Source-code Summarization of Java Methods Using Control-Flow Graphs

A thesis submitted to the
College of Graduate and Postdoctoral Studies
in partial fulfillment of the requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Michael Sheleme Beyene

Permission to Use

In presenting this thesis in partial fulfillment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Disclaimer

Reference in this thesis to any specific commercial products, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement, recommendation, or favoring by the University of Saskatchewan. The views and opinions of the author expressed herein do not state or reflect those of the University of Saskatchewan, and shall not be used for advertising or product endorsement purposes.

Requests for permission to copy or to make other uses of materials in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building, 110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan S7N 5C9 Canada

OR

Dean
College of Graduate and Postdoctoral Studies
University of Saskatchewan
116 Thorvaldson Building, 110 Science Place
Saskatoon, Saskatchewan S7N 5C9 Canada

Abstract

Source-code summarization aims to generate natural-language summaries for software artifacts (e.g., method and class). Various research works showed the use of text-retrieval-based techniques, heuristic-based techniques, and data-driven techniques for source-code summarization. In data-driven techniques, researchers used a sequence of source-code tokens and other representations of source code (e.g., application programming interface (API) sequences and abstract syntax tree (AST)) as an input to source-code summarization models. According to the current published literature in source-code summarization, researchers have not explored the use of a sequence extracted from control-flow graph that shows a contextual relationship between program instructions based on control-flow relationships for source-code summarization models. In this work, we employ control-flow graph representations to increase the prediction accuracy of a bi-directional long-short term memory (LSTM) source-code summarization model in terms of describing the functionality of Java methods. We use an attention-based bi-directional LSTM sequence-to-sequence model to show the use of linearized control-flow graph sequences alongside a sequence of source-code tokens. We compared our model with the current state-of-the-art and with or without a linearized control-flow graph. We created a source-code summarization dataset to train and evaluate our approach and conducted expert and automatic evaluations. In the expert evaluation, the participants gave rating for summaries generated by each model in terms of correctly describing the functionality of a Java method. Our models outperformed the state-of-the-art in terms of the mean average-rating. Also, the expert evaluation showed us the model benefit from the structural information. In the automatic evaluation, we found that the use of control-flow graphs does not increase the prediction accuracy of a bi-directional LSTM model in terms of BLEU score compared to a bi-directional LSTM model that does not use control-flow graphs. However, we found our source-code summarization approach that uses a control-flow graph as an additional representation better than encoding AST in graph neural networks. Overall, we improved the state-of-the-art for method summarization with our models that take sequence of method tokens with and without a control-flow graph.

Acknowledgements

First, I would like to thank my supervisors Prof. Christopher Dutchyn and Prof. Kevin Schneider, for their academic and financial support for this graduate thesis work. They helped me to grow as an academician and also as a person. I would also thank them for accommodating and supporting me during the difficult times during my study.

Second, I would also like to thank my lab mates and students that participated in my study as an expert to evaluate our approach.

Finally, I would like to thank my family and friends for their emotional support during my study.

To my family Babi, Geni, Sosi, and Mulu

Contents

List of Tables

List of Figures

List of Abbreviations

AST	Abstract syntax tree
API	Application programming interface
Bi-LSTM	Bidirectional Long-short term memory
GRU	Gated recurrent unit
LSTM	Long-short term memory
RNN	Recurrent neural network
SSA	Single static assignment
SWUM	Software word usage model

1 Introduction

Internal documentation (source code and comments) is useful for understanding programs when making software changes [67]. The purpose of these changes can be to add new functionalities, adapt to new environments, and correct programming errors. In making a software change, programmers spend half of the allocated time on program understanding tasks [61], including gathering information from internal documentation (source code and comments), from user documentation, and from specification documents (e.g. Software Requirement Specification and Software Design Specification documents) [18]. The availability and adequacy of external documentation for program understanding depends on allocated resources and the software development methodology used (e.g. extreme programming) for a project [10]. For this reason, it is common to rely on internal documentation for program understanding tasks.

Fluri *et al* [20] conducted an empirical study on three open-source systems (ArgoUML, Azureus, and JDT Core) to analyze the practice of updating comments as changes made on the source code. They reported that the comment coverage for method declarations is less than 20% for all subject systems used in the study. This implies a need for an automated alternative to generate summaries that covers all the methods in a project.

Also, Kajko-Mattson [36] conducted a survey to analyze documentation practices when correcting bugs across 18 companies based in Sweden. He collected data from individual interviewees who were willing to give information about the corrective maintenance culture of their organization. He defined 19 documentation requirements and assessed each company based on the requirements. The requirements expect the documentation to be done at different artifact levels (e.g., method, class, and packages). Also, the documentation is expected to be correct, complete, and consistent. In general, the requirements enumerate conditions that must be satisfied by the documentation to aid corrective maintenance tasks. One of the key requirements was having company-wide guidelines for internal documentation to ensure the readability and maintainability of source code. According to the collected information, more than one-third of the participant companies do not have guidelines for internal documentation. Since developers have different writing styles, having no internal documentation guidelines may introduce inconsistencies in the internal documentation.

Source-code summarization systems generate summaries that can be used as internal documentation. The summaries are generated automatically and are current since new summaries can be generated as the source code changes. Also, the summaries can be generated for every method in a project, resulting in complete

```java
//this method return the application url path
public String getPath() throws MalformedURLException {

    URL url=getURL(this.url);

    String path=url.getPath();

    if (path.length() == 0)
        path=null;

    return path;
}
```

Figure 1.1: Caption summary example

comment coverage. Due to this, we will focus on generating caption summaries for source-code method. Figure 1.1 shows an example method with a caption summary on the top of the method. Our approach generated the string: "returns path url as string" for the method shown in the figure.

Source-code summarization systems are designed to generate natural-language summaries for a given source-code artifact (block, method, class, and packages) [22]. Researchers have shown different approaches to summarize source code. These approaches can be grouped into three categories: *text-retrieval*, *heuristic*, and *data-driven-based* source-code summarization. Text-retrieval-based approaches use information-retrieval techniques to select and output tokens that can represent a given source-code artifact. Heuristic-based approaches use a combination of program analysis and natural-language-processing techniques to summarize source code. Data-driven-based approaches use deep learning to generate summaries for a given source-code artifact. We discuss the details of each summarization approach in Chapter 2 (Background) and Chapter 3 (Experimental Design).

There is a disadvantage associated with text-retrieval and heuristic-based approaches: the quality of summaries generated by these approaches depends on the name of identifiers and method names used in the subject source-code artifact. They require expressive names that provide meaning by themselves or make sense when fit into a pre-defined template. For this reason, data-driven approaches are becoming dominant in the area of source-code summarization.

Data-driven summarization approaches differ based on the selected input representation and model architectures used to implement the model. Mainly, researchers use models that are sequence-to-sequence models. These models are designed to transform an input source-code artifact represented as a sequence of tokens into an output sequence like a summary. In a source-code summarization context, they transform a sequence created from tokenized methods and/or other selected input representation to a source-code summary.

Recent work in data-driven source-code summarization explore the benefit of encoding structural information such as AST (abstract syntax tree) in the source-code summarization models [4, 28, 41]. These representations have been used as input in addition to method token sequences and by themselves. The results show source-code summarization models benefit from structural information. We discuss these studies in Sections 3.1 (Input Representation) and 3.2 (Model Architecture).

The success of AST in source-code summarization models indicates a more detailed structural representation would seem likely to support the generation of more accurate descriptive summaries for methods.

Following this, we select a more detailed structural representation that captures the control-flow relations between blocks of instructions inside a method. This representation is called *control-flow graph*. We construct this graph from *single-static-assignment (SSA)* form of a program, which is an intermediate representation in which a variable has only one definition. This allows us to easily connect the definition and use of variable in constructing the control-flow graph.

In this work, we employ control-flow graph representations to increase the prediction accuracy of a bi-directional long-short term memory (LSTM) source-code summarization model for describing the functionality of Java methods.

We represent a Java method in two ways: as a linearized control-flow graph and as sequence of method tokens. First, we transform the SSA form of a program into a linearized control-flow graph. Second, we tokenize the method into a sequence of method tokens. The sequence of method tokens is created by removing punctuation and separating keywords and identifiers by white space and delimiters. We add this representation to encode identifier names replaced with generic names in transforming a program to an SSA form.

We adapt a multi-encoder sequence-to-sequence architecture to encode our representations from LeClair *et al.* [41]. We use two bi-directional LSTM encoders to encode method tokens and control-flow graphs. Bi-directional LSTM reads the input both from beginning to end and from end to beginning. On the other hand, uni-directional LSTM reads the input from beginning to end or from end to beginning. For the summaries, we use a uni-directional LSTM decoder. To assist the decoder in focusing on relevant tokens, we use a simple and effective attention mechanism by Luong *et al.* [48]. We call this model *method_ cfg*. We discuss details in Section 2.2 (Sequence-to-Sequence models).

To demonstrate our success at increasing accuracy, we compare our model with two models using an expert evaluation and an automatic evaluation. We implemented the first model, a bi-directional LSTM

model that only has one encoder that takes a sequence of method tokens as input. We call this model the *method_ only* model. The second model is the current state-of-the-art by LeClair *et al.* [40]. We call this model *codeGNN*. In this way, we can appraise the improvement available from control-flow graph encoding and recognise the improvement over the current best result.

Each evaluation serves a different purpose. The automatic evaluation is an accepted form of evaluation and is used in related work [28, 31, 40, 41]. We used BLEU score [54] as an evaluation metric. BLEU measures the n-gram precision of the predicted summaries according to the ground truth. Even though the BLEU score is a popular evaluation technique in source-code summarization, it has potential flaws. BLEU score is sensitive to the quality of a dataset and doesn't handle paraphrases. By "the quality of the dataset", we mean a dataset with multiple and verified reference translations. This kind of dataset is difficult to create, especially in deep learning approaches that require large datasets. The expert evaluation is included as a sanity check for the automatic evaluation. Also, it helps us evaluate the generated summaries based on purpose, which is correctly describing the functionality of a given method. Papieneni *et al.* showed that BLEU correlates with human judgement; with this, a higher BLEU score expected a higher value expected to receive a higher evaluation score from human evaluators.

As a preview of our results, in the expert evaluation, 26 students: 19 undergraduate, 3 M.Sc., and 4 P.h.D candidates judged summaries generated by our model and the two baselines. Our results show that method_cfg received a higher mean average-rating compared to the two baselines, with a mean average-rating of 3.44 out of 5. The method_only model received 3.42, and LeClair *et al.'s* model received 1.46. Based on our review, summaries generated by our models are not accurate enough to completely replace human involvement in writing summaries for source-code methods. But, they can be used in a setting where humans validate the generated summaries for correctness.

In the automatic evaluation, we conduct 10-fold cross-validation on a dataset split at the project level to produce training, tesitng, and validation sets. A project-level split helps us evaluate the models with a test set constructed from projects not seen by the model during training. In the evaluation, the method_only model outperformed our model in eight out of 10 folds. In turn, method_cfg outperformed the state-of-the-art in seven out of 10 folds. Also, on a dataset by LeClair *et al* method_only got 17.53 while codeGNN got 19.93.

Our evaluations show that our approaches improved the current state-of-the-art over our dataset. Overall, the model that does not use control-flow graphs outperformed the model with control-flow graphs.

1.1 Contribution of the work

This study has the following contributions:

- We constructed a source-code summarization dataset by extracting high-quality projects from Martins *et al.* [49] dataset and provide method token sequences annotated with summary, control-flow graphs, and ASTs in XML. Others can use this dataset for future work. You can access the dataset using this link.
- We improved the state-of-the-art (codeGNN) method summarization on our dataset with (method_cfg) and without (method_only) control-flow graphs. Also, our method_only model got marginally inferior compared to the state-of-the-art (codeGNN) result on a dataset by LeClair *et al.*
- We sanity-checked the automatic BLEU score evaluations using humans experts.

1.2 Overview

We organise the remaining part of this thesis as following:

- **Chapter 2: Background**: provides a background to the reader in order to understand the design, implementation and evaluation of our experiments.
- **Chapter 3: Experimental Design**: presents details of our dataset, representation, model architecture, and evaluation with respect to other related works.
- **Chapter 4: Implementation**: explains the implementation details of our approach.
- **Chapter 5: Evaluation**: discusses the results of our experiments.
- **Chapter 6: Summary** : provides summary of this thesis document.

2 Background

This chapter provides a brief introduction to topics in order that the reader knows the specific variations and details, we rely on for the rest of the document. Section 2.1 briefly describes source-code summarization systems and discusses approaches used to design and implement the systems. We discuss the family of models we chose to implement our system called sequence-to-sequence models in Section 2.2. Next, we describe our automatic-evaluation approach in Section 2.3. Finally, we explain the representation we used to train our models in Section 2.4

2.1 Source-code Summarization

Automatic summarization systems aim to select (extract) and/or generalize (abstract) a concise output text from a given source document [35]. The output of these systems (summary) is expected to capture relevant information in the source document. Also, the output summaries must accurately describe or represent the original document.

Based on the content of the output generated by summarization systems, there are two types of automatic-summarization approaches: *extractive summarization* and *abstractive summarization*. An extractive-summarization system creates a summary by selecting relevant terms from the source document [23]. An abstractive-summarization system outputs a summary that provides a high-level description of the source document. The difference between the two approaches is most visible in their output. An output of an abstractive-summarization system may include terms that do not exist in the source document. On the other hand, an output of an extractive-summarization system does not contain terms that do not exist in the original document.

Source-code summarization resembles text summarization, and so similar techniques apply.

Source-code summarization systems aim to generate summaries for source code in different artifact levels (block, method, class, and package). Haiduc *et al.* [22] describes summaries generated by automatic source-code summarization systems as textual descriptions of source-code artifacts that help developers understand source code and capture precisely. According to the requirements stated by Haiduc *et al.*, the generated summaries are expected to be concise and reflect the intent of developers. In other words, source-code summarization systems are expected to produce a short summary that correctly describes a given source-code

artifact.

Researchers studying source-code summarization approaches employ techniques that can be grouped into three categories: *text-retrieval-based techniques*, *heuristic-based techniques*, and *data-driven techniques*. Here, we discuss text-retrieval- and heuristic-based work covered by the literature survey of Zhu and Pan [72]. We further filtered the list of work by selecting research that studied method-level automatic source-code summarization. Our work also comes under method-level automatic source-code summarization.

Extractive techniques have been explored and show some potential. We start by detailing text-retrieval techniques. Text-retrieval techniques are extractive. But, summaries generated by latent semantic indexing (LSI) and topic modeling approaches may include terms from the text corpus that are not inside the source document. Eddy *et al.* [17]called this light-weight abstractive.

Text-retrieval-based source-code summarization systems adopt information-retrieval techniques to extract relevant tokens (keywords and identifiers) that represent a source-code artifact [33]. In the literature, we found four text-retrieval methods used by researchers to summarize source code. These are *lead*, *vector-space model (VSM)* [60], *latent semantic indexing (LSI)* [14], and *hierarchial pachinko allocation model (HPAM)* [52].

Lead-summarization technique assumes the initial terms of a source document captures the context of the document. This is parameterized N, a positive integer selected by the user [23]. Lead summaries are created by extracting the first N terms from the source document. For example, the lead summary of a method can be tokenized signature of the given method.

VSM and LSI start with creating a term-document matrix from a text corpus but differ in their use of the matrix.

The columns of a term-document matrix represent documents (e.g., methods and sentences) inside the corpus, and the rows represent terms in the corpus. Each cell of the matrix contains a weight that shows the relevance of the term to the document and the corpus depending on the selected weighting scheme. In natural language summarization, *log*, *tf-idf (term frequency-inverse document frequency)* [34], and *binary-entropy* perform best for both LSI and VSM [23]. We look at each weighting scheme separately.

- Log scales the frequency of terms inside a document logarithmically.
- Tf-idf determines the relevance of the term in both the document and the corpus. The equation to compute tf-idf is shown in Equation 2.1. Term-frequency (tf) is the frequency of term t in a document.

Inverse document frequency (idf) is the inverse of the document frequency, which the log of n (the number of documents in the corpus) divided by df the number of documents containing term t. The product of term frequency and inverse document frequency gives the final tf-idf weight of the term. The idf factor makes the tf-idf value for rare words high since the documents containing the term t (df) are small; on the contrary, it makes the tf-idf value of frequently-occurring words low since the df value is high [59].

$$tf - idf(t) = tf(t) \cdot idf(t),\ idf(t) = \log\left(\frac{n}{df} + 1\right) \tag{2.1}$$

$$G_i = 1 - \sum_j \frac{p_{ij}\log(p_{ij})}{\log(n)},\ p_{ij} = \frac{t_{ij}}{g_i} \tag{2.2}$$

- Binary entropy is a product of a local binary weight and a global entropy weight [64]. The local weight is zero if the term does not exist in the document and one if it exists. The equation for the global weight G_i is shown in Equation 2.2. In the equation, t_{ij} is the frequency of the term i in the document j, g_i is the frequency of the term in the corpus, and n is the number of documents in the corpus. The G_i value for frequently occurring terms is small compared to rare terms, and this makes the weighting scheme give higher relevance to rare words that determine the meaning of a document.

Haiduc *et al.* [23] conducted a participant study that compared the weighting schemes (log, tf-idf, and binary entropy) for source-code summarization using VSM, LSI, and the combination of lead and VSM. The participants gave the highest score to the term-document matrix weighted using tf-idf in LSI and the combination of Lead+VSM text-retrieval approaches. The binary entropy weighting scheme got the highest score in VSM text-retrieval approach.

Now, we describe individual text-retrieval techniques.

VSM creates a summary by selecting the first N terms from a document in a term-document matrix with terms sorted based on weight. N is a positive integer set by the user.

In contrast to that simple approach, LSI uses singular value decomposition to derive a matrix that shows the hidden relationship between terms and documents of a term-document matrix. LSI creates a low-rank approximation of the original term-document matrix: It factorizes the term-document matrix using singular value decomposition, as shown in Equation 2.3. The singular value decomposition decomposes the original matrix into three matrices U, S, and V. First, The low-eigenvalues entries in the diagonal

$$A = USV \tag{2.3}$$

matrix S will be removed, and we take the rank (R) of the left-over matrix. Second, the first R columns will be taken from the first matrix U. Also, the first R rows of matrix V will be taken. Then, we multiply the resulting low-rank matrices U, S, and V. Finally, we index N terms of a document from the resulting term-document matrix that is similar to the document being summarized. N is a positive integer set by the user.

Last, topic models are a set of algorithms designed to discover hidden thematic structures (topics) in unstructured text [8]. HPAM is a topic-modeling approach built upon the latent Dirichlet allocation (LDA model. LDA models a document as a probability distribution of topics and each topic as a probability distribution of words. HPAM models a text corpus into a hierarchy of topics and words. The hierarchy is a directed acyclic graph created from word-subtopic probability distributions, subtopic-super topic probability distributions, and super topic-document probability distributions. In the graph, each node is connected to lower-level nodes in the hierarchy, which is the document is connected to super-topics, super-topics are connected to sub-topics, and sub-topics are connected to words. Also, each super-topic and subtopic has a probability distribution over words in the document. During retrieval, HPAM selects the most likely subtopic for a given document. The selected sub-topic has it is own word distribution over the vocabulary. Then, the top N terms in the word distribution of the sub-topic are selected from the sub-topic to create a summary. N is a positive integer set by the user.

Researchers employed these methodologies to create summaries for source code.

First, Haiduc *et al.* [23] conducted a study that compares the quality of extractive-summary generated by lead, VSM, and LSI for Java methods and classes. They used human evaluation to compare summaries generated by each technique. Among these techniques, the study participants gave the highest score for summaries generated by the lead summarization technique. They also showed the union of lead and VSM summarization techniques a better summary than summaries generated using only lead summarization.

Second, Eddy *et al.* [17] extended and replicated the work of Haiduc *et al.* [23] by conducting a user study that compared summaries generated by lead, VSM, and HPAM. They also compared the combination of lead and VSM with the combination of lead and HPAM. Their result supported Haiduc *et al.*, where the lead summarization technique received the highest score. Among the combined techniques, lead, and VSM a higher rating than lead and HPAM.

Both, Haiduc *et al.* and Eddy *et al.* asked the study participants to judge the quality of summaries

generated by the text-retrieval approaches we discussed above. Overall, among text-retrieval summarization approaches, the union of VSM and lead summarization techniques yield a better term-based extractive summary of classes and methods compared to lead, VSM, LSI, and HPAM. Both studies used two subject systems (atunes - a media player and arts of illusion - a graphics software) to evaluate their approach. To the best of our knowledge, there were no follow up studies conducted to both to replicate and extend the results of Haiduc *et al.* and Eddy *et al.* We recommend extending and replicating their work in a more diverse set of project types to generalize their result.

Next, we discuss heuristic-based summarization techniques. Like text-retrieval-based techniques, heuristic-based techniques are effective.

Heuristic-based summarization techniques are a rule-based transformations of source-code artifacts to natural-language summaries. They combine program analysis and natural-language-processing techniques to summarize source-code artifacts. Most work under this category operate by extraction of terms from the given source-code artifact and place the extracted terms in a pre-defined template. Heuristic-based summarization techniques used for method-level source-code summarization comes under *software-word-usage-model-based* [25], *method-stereotype-based* [16], and *nano-pattern-based* [62] source-code summarization techniques. Studies discussed under heuristic-based summarization were evaluated in a qualitative way; but user-studies have shown them to be effective.

The software-word-usage model (SWUM) was introduced for query reformulation. SWUM is a method that generates noun phrases (e.g, key pressed), prepositional phrases (e,g, to byte array), and verb phrases (e.g, get item) from method and field signatures (method calls) for query reformulation to locate features in source code. In source-code summarization, SWUM is used to extract and identify parts of speech from a given method or field signature (e.g., for `saveImage()`, save is the verb, and image is the object). Then, extracted tokens will be placed in a predefined template to create sentences out of a statement (e.g., `saveImage()` gets transformed into "this method saves an image"). Finally, the output summary will be created by concatenating sentences formed from the important statements. We discuss individual approaches of statement selection later in this section.

Method stereotypes are high-level descriptions of the responsibility of a method inside a class [16] (e.g., accessor (getters), mutators (setters), and creational (constructors)). Method-stereotype identification tools use static analysis to extract information from methods. Then, a set of heuristics are applied to determine method-stereotypes from the extracted -information. Method stereotypes are used in source-code summarization to select a pre-defined template to include the extracted information into a summary [1].

Nano-patterns [62] are similar to method-stereotypes, except they do not give us the responsibility of methods in a class. They capture properties exhibited by source-code methods, including the way they are called (e.g., no-params: a method that takes no arguments and no-return: a method that mutates state or logs results but does not return a value), by the way they interact with objects. (e.g., object creator: constructs new object and field reader: read values from object fields), by control flow inside the method (e.g., straight-line: no branches in method body and looping: performs repetitive tasks), and by data flow inside the method (e.g., local-reader: read values from local variables and local-writer: write values of local variables). Static analysis, including control-flow and data-flow analysis, is used to determine a nano-pattern for a given method [57]. In source-code summarization, nano-patterns are used to select a template to embed the extracted information from a method.

Next, we discuss how these techniques are applied for source-code summarization.

Sridhara *et al.* [63] used SWUM to extract and identify parts of speech from source-code statements that are selected to represent a given method. The statements are selected by applying heuristics that refer to their location in source code (return statements), control-flow dependence (control statements such as if and while statements), and relation with other statements (statements that execute similar action). Next, their approach selects the right template and converts each statement to a sentence by placing the extracted tokens into a pre-defined template. McBurney and McMillan [51] improved the work of Sridhara *et al.* by including method contexts in addition to sentences converted from method signature and body using SWUM. The method context is method calls within the method and methods that called the method being summarized. They used the page-rank algorithm [53] to rank method contexts. Then, they placed the extracted tokens and method contexts in a pre-defined template.

Abid *et al.* [1] showed the application of method stereotypes for source-code summarization. In their approach, method stereotypes are used to describe the role of a method inside a class. In addition, method stereotypes are used to determine the contents of the final summary. To generate summaries, they placed tokens extracted using static analysis into a pre-defined template selected based on method stereotypes.

Rai *et al.* [57] explored the use of nano-patterns for source-code summarization. They used static analysis to determine nano-patterns of methods. After identifying the nano-patterns, they place tokens extracted from methods into a pre-defined template prepared for each nano-pattern. In the case of multiple nano-patterns, they concatenate summaries generated for each nano-patterns of the subject method. This approach is similar to Abid *et al.*, but they used nano-patterns instead of method stereotypes.

McBurney and McMillan compared their approach with Sridhara *et al.* by conducting a user study. The user study had 12 participants: nine computer science and engineering graduate students at the University

of Notre Dame and three programmers. They sampled 20 methods from six projects. They showed their approach generates a better summary than the approach by Sridhara *et al.* The evaluation metrics were accuracy, conciseness, and generation of summaries with contextual information that helps programmers better understand the generated summaries.

Similarly, Rai *et al.* compared their approach with Sridhara *et al.*, and McBurney and McMillan with a user study. The user study had 40 participants: 36 undergraduate and four graduate students of the Department of Computer Science and Engineering at PDPM Indian Institute of Information Technology, Design and Manufacturing. For the evaluation, they sampled 15 summaries from six projects. The evaluation criteria were correctness, completeness, non-redundancy, and conciseness of summaries generated by each approach. Completeness is a subjective evaluation that measures if a summary had all potentially required facts. Their results show that their approach is better in generating correct, non-redundant, and complete summaries.

Abid *et al.* conducted an initial manual evaluation using undergraduate students, but the evaluation details such as the number of participants, qualifications of participants, and the quantitative result of the evaluation were not discussed in the paper. Also, they did not compare their approach to other related works. We included the details of our expert evaluation that covers participant detail, quantitative result, sampling strategy, and model details in Section 3.4.2 and Section 5.1.

Text-retrieval- and heuristic-based summarization approaches also have some disadvantages. Text-retrieval- and SWUM-based techniques rely on the quality of identifiers and method names [1]. Method-stereotype-based and nano-pattern-based summarization approaches do not generate a concise summary that describes the functionality of the method; instead, they give a generalized description of a method based on method stereotype.

Works under data-driven summarization approaches use deep learning to generate natural-language summaries for source-code artifacts. As these resemble our effort, we defer the discussion of these works on the next chapter (experimental design) in Section 3.1 (input representation) and Section 3.2 (model architecture).

2.2 Sequence-to-Sequence Models

Sequence-to-sequence models are a family of deep-learning models that transform an input organized as a sequence (such as a sequence of tokens) to an expected output sequence [66] such as a code summary. These models are applied in neural machine translation, video captioning, text summarization, source-code summarization, and other related tasks.

As a deep learning model, sequence to sequence models do two types of computations: forward propagation and backpropagation. Forward propagation computes the model output by transforming the input using consecutive layers processing units (hidden layers). On the other hand, backpropagation iteratively updates the training parameters to minimize the prediction error of the model [58].

A sequence-to-sequence model aims to minimize the loss of the cost function (categorical cross-entropy). Categorical cross-entropy computes the difference between two probability distributions: the ground truth and the predicted distribution. Equation 2.4 shows the categorical cross-entropy equation. When we generalize it, the goal is minimizing the cross-entropy c by decreasing the difference between the target one-hot encoding vector y and the predicted distribution vector x with a size of target vocabulary (number of classes) v. In machine learning, categorical cross-entropy is used to compute the loss of multi-class classification. In this case, the final layer of a sequence-to-sequence function predicts a probability distribution over the vocabulary of the output sequence based on the context vector and output of the previous time-step.

$$c = -\sum_{i}^{v} y_i log(x_i) \tag{2.4}$$

Objective function

In this work, we use sparse categorical cross-entropy. In sparse categorical entropy, the true labels are represented as an integer rather than one-hot encoding [39]. One hot encoding is a mechanism of representing data in a bit vector the size of categories (in our case, the size of vocabulary), where all bits are 0 except one bit is 1. For example, for two categories, we will have "10" for the first category and "01" for the second category. In integer representation, it will be 1 for the first category and 2 for the second category. When we use one-hot encoding to represent our labels, we use categorical cross-entropy.

Table 2.1: Operation done by each sequence processing model cell type

Model	**Cell**	**Output**
RNN	Equation 2.5	Equation 2.5
LSTM	Equation 2.6 - 2.11	Equation 2.10 and Equation 2.11
GRU	Equation 2.14 - 2.17	Equation 2.17

Sequence-to-sequence models have two components: an encoder and a decoder. The encoder converts an input sequence to a context vector, and the decoder learns to predict an output sequence from the context

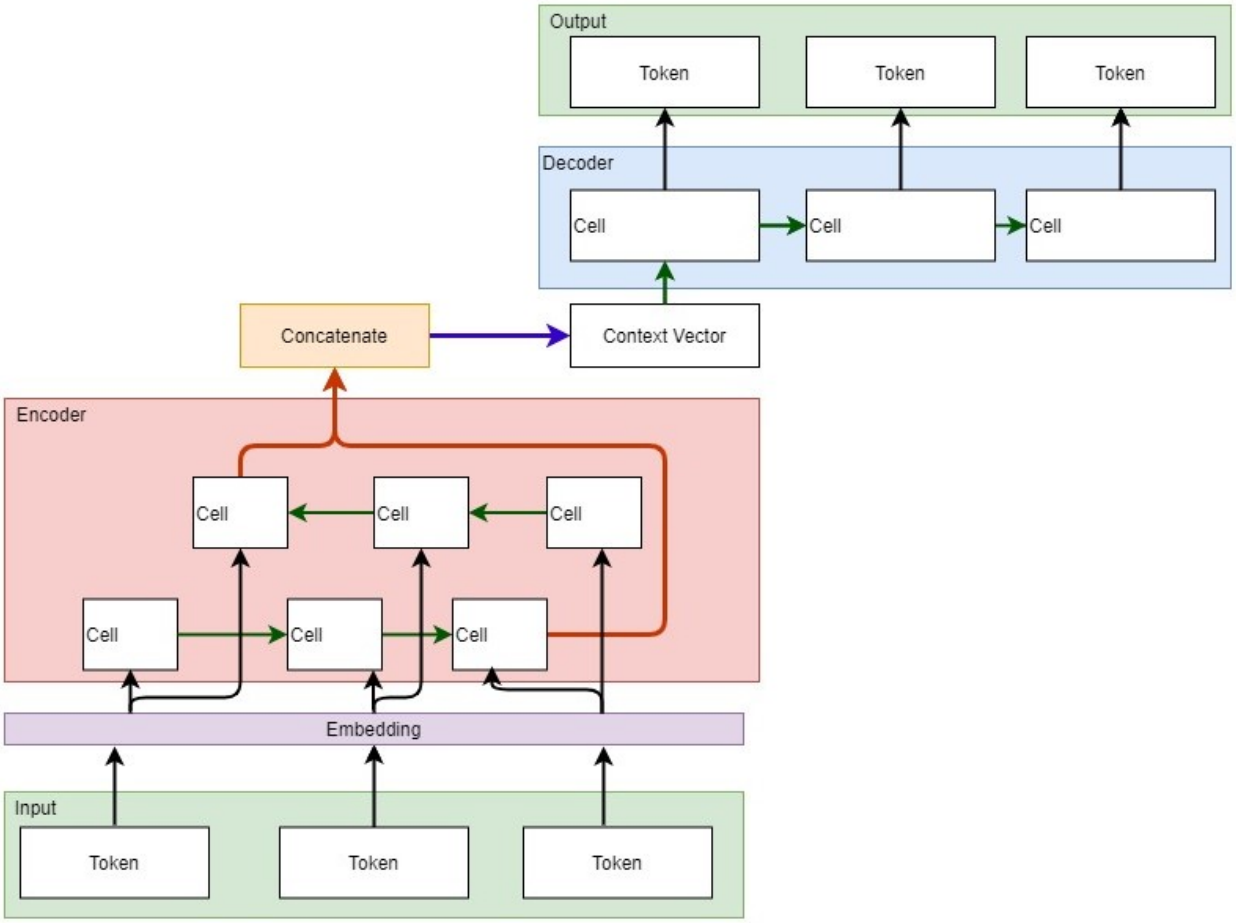

Figure 2.1: Bi-directional sequence-to-sequence model

vector [12]. Figure 2.1 shows a sequence-to-sequence model with encoder-decoder architecture. As shown in the figure, the encoder has cells that accept and process an input and a hidden state from the previous cell. The decoder also contains cells that predict output based on the context vector and the state of the previous cell.

Here we discuss three types of sequence models (models that process sequential data) based on cell type: Recurrent neural network (RNN) [70], Long short-term memory(LSTM) [27], and GRU [12]. Each of the three types fit into the structure in Figure 2.1. They differ computations done in the cells and the type of information carried by arrows that connect each cell. We colour coded the arrows based on operations. The black arrows carry the input and output of the cells. The green arrows carry information from cell to cell. The orange arrows show the transfer of the last hidden state computed in each direction to concatenation layer that stacks the hidden states horizontally. The purple arrows show the creation of the context vector after concatenating the hidden states from each direction. Also, in the figure, we see an embedding layer that transforms the input into real-valued vectors that capture the semantic of the input in relation to its neighboring contexts. We summarize the operations done by each cell type in Table 2.1. The table contains

the list of equations associated with each sequence model cell type.

RNN is a type of neural network designed to process sequential data [65]. We can see the application of RNN in speech recognition, music generation, text generation, and similar problem domains. As shown in the figure, each cell is a time step to process an input and a hidden state from the previous time step. In the encoder-decoder architecture, the RNN encoder predicts an output at the final time step, and the decoder uses the prediction to predict an output on each time step. The equation to compute a hidden state in a single RNN cell is shown in Equation 2.5. In the equation, hidden state h_t is computed by adding bias (b_h) to the sum of the product of the weight matrix W_h and the previous hidden state h_{t-1}, and the product of W_h and the current input x_t.

$$h_t = \sigma(W_h h_{t-1} + W_x x_t + b_h) \tag{2.5}$$

Computation inside a single RNN cell

$$o_t = \sigma(W_{oh} h_{t-1} + W_{ox} x_t + b_o) \tag{2.6}$$

$$i_t = \sigma(W_{ih} h_{t-1} + W_{ix} x_t + b_i) \tag{2.7}$$

$$f_t = \sigma(W_{fh} h_{t-1} + W_{fx} x_t + b_f) \tag{2.8}$$

$$c'_t = tanh(W_{ch} h_{t-1} + W_{cx} x_t + b_c) \tag{2.9}$$

$$c_t = i_t \otimes c'_t + f_t \otimes c_{t-1} \tag{2.10}$$

$$h_t = o_t \otimes tanh(c_t) \tag{2.11}$$

$$\sigma(z) = \frac{1}{1 + exp(-z)} \tag{2.12}$$

$$tanh(z) = \frac{exp(z) - exp(-z)}{exp(z) + exp(-z)} \tag{2.13}$$

Computations in a single LSTM cell

The problem with RNN is that during training, the gradients being backpropagated through time to update parameters may vanish [27]. Especially, these problems tend to persist in processing longer sequences [56]. This makes it hard for the model to learn long-term relationships between inputs in distant time steps. LSTM and GRU are designed to solve to these problems.

LSTM is designed to capture long-term dependency within an input sequence. Long-term dependency

is maintained by transferring a cell state in addition to a hidden state. Each LSTM cell has three gates to process an input and states received from the previous time step. These gates are input (update) gate, forget gate, and output gate. The input gate processes an input of the current timestep. The forget gate decides whether to keep or discard a cell state received from the previous time step. The output gate computes the hidden state (prediction) at the current time step.

Equations 2.6 to 2.13 show computations inside an LSTM cell. In equations 2.6 to 2.8, we can see that the gates that control the output of the timestep both cell state and hidden state use the sigmoid function. The sigmoid function outputs numbers between 0 and 1. The element-wise multiplication between the output of the gates and cell states (previous and candidate) decides whether to replace or keep the previous cell state. Also, it decides the output hidden state that will be passed to the next hidden state.

- o_t is the output gate. It is a sigmoid function of a computation done by adding a bias (b_o) to the product of a weight matrix (W_{oh}) and the previous hidden state (h_{t-1}) to the product of a weight matrix (W_{oX}) and the current input (x_t).
- i_t is the input gate. It is a sigmoid function of a computation done by adding a bias (b_i) to the product of a weight matrix (W_{ih}) and the previous hidden state (h_{t-1}) to the product of a weight matrix (W_{iX}) and the current input (x_t).
- f_t is the forget gate. It is a sigmoid function of a computation done by adding a bias (b_f) to the product of a weight matrix (W_{fh}) and the previous hidden state (h_{t-1}) to the product of a weight matrix (W_{fX}) and the current input (x_t).
- c'_t is the candidate cell state to update the previous cell state (c_{t-1}). It is a hyperbolic tangent (tanh) function of a computation done by adding a bias (b_c) to the product of a weight matrix (W_{ch}) and the previous hidden state (h_{t-1}) to the product of a weight matrix (W_{cX}) and the current input (x_t).
- c_t is the current cell state that will be forwarded to the next time step. It is computed by adding element wise product of the input gate output (i_t) and candidate cell state (c_{t-1}) to the element wise product of forget gate output (f_t) and the previous cell state (c_{t-1}).
- h_t is the current hidden state. It is the element-wise product of output of the output gate and tanh function of the current cell state.

Gated recurrent unit (GRU) is also designed to capture long term dependencies. To capture long-term dependence GRU also have a mechanism to keep or discard a cell state based on relevance. Unlike LSTM, GRU has an input (update) gate and a forget (reset) gate. This makes it computationally efficient. In addition, GRU does not transfer cell state in addition to hidden state, it only controls the value of the hidden state using the update

$$u_t = \sigma(W_{uh}h_{t-1} + W_{ux}x_t + b_u) \tag{2.14}$$

$$r_t = \sigma(W_{rh}h_{t-1} + W_{rx}x_t + b_r) \tag{2.15}$$

$$h'_t = \tanh(W_h(r_t \otimes h_{t-1}) + W_h x_t + b_h) \tag{2.16}$$

$$h_t = (1 - u_t) \otimes h_{t-1} + u_t \otimes h'_t \tag{2.17}$$

Computations in a single GRU cell

and forget gate. Equations 2.14-2.17 show operations computed by a GRU cell. u_t is the update gate computed from previous hidden state h_{t-1} and current input x_t. The reset gate r_t is also computed form h_{t-1} and x_t. The candidate hidden state h'_t is tanh function of the sum of element-wise product of r_t and h_{t-1} multiplied by weight matrix W_h, and product of W_h and x_t plus the bias b_h. Finally, the output hidden state h_t is computed using the output of the update gate u_t, the previous hidden state h_{t-1} and the candidate hidden state h'_t.

In standard sequence-to-sequence models, the encoder predicts the context vector at the final time step, and the decoder learns to predict an output sequence based on the context vector. This approach raises a problem in encoding large sequences into a fixed-length context vector [6]. Bahdanau *et al.* introduced a mechanism commonly known as attention that informs the decoder to align (focus) to an encoder state at a time while predicting an output on each time step.

Equations from 2.18 to 2.20 show how a context vector is computed using the attention mechanism by Bahdanau *et al.*

- e_{ij} is the alignment score. It is computed by passing previous decoder state s_{i-1} and encoder state h_j to a feed-forward neural network a with tanh activation function.
- α is the alignment weight computed by passing the alignment score e_{ij} to a softmax layer. This converts the alignment score to a probability distribution which informs the decoder to align to a specific encoder state h_j with respect to the previous decoder hidden state s_{i-1} to predict the current output s_i.
- c_i is the context vector. It is the summation of the product of the alignment weight α with each encoder hidden state $(h_1, ..., h_{Tx})$ where T_x is the length of the input sequence.

In our study, we used another attention mechanism introduced by Luong *et al.* (Luong-attention) [48]. We have three reasons for choosing Luong-attention. First, we adapted the multiple encoder architecture of LeClair *et al.* [41] which also used Luong-attention. Also, we compared our final work with their work that used Luong-attention. Second, Luong-attention is introduced as a simple and effective alternative to incorporate attention in neural-machine translation-models. With this attention mechanism, it is simple and

$$e_{ij} = a(s_{i-1}, h_j) \tag{2.18}$$

$$\alpha = \text{softmax}(e_{ij}) \tag{2.19}$$

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j \tag{2.20}$$

$$softmax(e_{ij}) = \frac{e_{ij}}{\sum_{k=1}^{T_x} exp(e_{ik})} \tag{2.21}$$

Attention by Bahdanau *et al.*

intuitive to create a context vector created from the output of the decoder and multiple encoders.

$$e_{ij} = s_i^T.h \tag{2.22}$$

$$\alpha = \text{softmax}(e_{ij}) \tag{2.23}$$

$$c_i = \alpha.h \tag{2.24}$$

Attention mechanism by Luong *et al.*

Equations 2.22 to 2.24 shows the mechanism to compute the context vector c_i using Luong-attention mechanism. The difference between Bahdanau *et al.* is the mechanism used to compute the alignment score e_{ij} and context vector c_i. The vector e_{ij} is the product of the transpose of the current decoder hidden state s_i and the encoder hidden states h. The context vector is the product of the alignment weight α and the encoder hidden states h.

In this study, we used a bi-directional LSTM with an attention mechanism by Luong *et al.* Bi-directional LSTM uses two LSTM layers to encode an input. The first layer is a forward LSTM which reads the sequence from start to end. The second LSTM reads the sequence in a reverse order from end to start. Then, the output of both LSTM layers will be concatenated and passed to the attention layer and the decoder.

2.3 BLEU Score

BLEU score is a neural machine translation evaluation metric [55]. BLEU measures the n-gram precision of the predicted output with respect to the ground truth. It computes a weighted average of variable length phrase matches against reference translation. We can measure precision by dividing the number of terms in the prediction that exists in the reference translation by the number of predicted terms.

Consider the following example:
Prediction: like like programming
Reference: i like programming

The uni-gram precision of the predicted output in this example is 3/3, which is 1. But, if we see the output, the model predicted the term "like" twice. To counter this problem, BLEU measures modified n-gram precision of the predicted output. The modification is BLEU sets the count of a matched term or phrase to the number of occurrences of the term or phrase in the reference translation. In our example, the term "like" only occurs once in the reference translation, so the modified uni-gram precision becomes 1/3, and it gives the term "programming" 1/3, which makes the total unigram precision 2/3. For bi-gram precision, we have two bi-grams "like like" and "like programming". Since, we have "like programming" in the reference, the bi-gram precision becomes 1/2. The tri-gram and tetra-gram precision is zero.

BLEU outputs a single number between zero and one after computing the n-gram precision for various n-gram sizes (usually 1-4). N-gram is an ordered sequence of n tokens in a sequence (e.g, sentence). In our results, we scaled the BLEU score to 100 for clarity. The number is a weighted average of the modified n-gram precision. Also, if the model prediction is shorter than the reference text, BLEU multiples the weighted n-gram precision by a constant called brevity penalty. The brevity penalty ensures that a model doesn't get an inflated precision score by predicting short outputs. A brevity penalty is computed by the equation shown in Equation 2.25, and BLEU score is computed using the formula shown in Equation 2.26.

In Equation 2.25, we compute r by summing the length of reference sentences that best match the predicted output, in case we have multiple reference sentences for each output. Otherwise, r is the sum of the reference sentences length in the test corpus, and c is the sum of the length of the predicted sentences. In Equation 2.26, n is the n-gram value ranging from 1 to N. The weight given to each n-gram is w_n, and the modified precision value for each n-gram is p_n.

Table 2.2 shows an interpretation of the BLEU score. We adopted the interpretation from documentation published by Google [9]. Still, this interpretation can be biased by the quality of the dataset, but we present a rough interpretation in the table to assist readers.

$$\text{BP} = \exp\left(1 - \frac{r}{c}\right), \; if \; c \leq r \tag{2.25}$$

$$\text{BLEU} = \text{BP}.\exp\left(\sum_{n=1}^{N} w_n \log(p_n)\right) \tag{2.26}$$

BLEU score

Table 2.2: BLEU score interpretation

BLEU score	Interpretation
<10	Almost useless
10-19	Hard to get the gist
20-29	The gist is clear, but has significant grammatical errors
30-40	Understandable to good translation
40-50	High quality translations
50-60	Very high quality, adequate, and fluent translations
>60	Quality often better than human

2.4 Control-flow Graph

Our representation will rely on control-flow graphs, so we briefly describe them here. A control-flow graph is a directed graph in which edges connect basic blocks of instructions inside a procedure [13]. A basic block of instruction is a linear sequence of program instructions having one entry point and one exit point [3]. These blocks of instructions are connected based on control-flow dependence. Figure 2.2 shows the control-flow graph of a simple program that adds or subtracts one from an input based on the input value. As we can see in the figure, we can trace possible execution paths by following the edges that connect the blocks of instructions from the entry point to the exit point of a program. One component of this analysis is tracing the definition- and use-site of variables [3]. Such a type of analysis becomes simpler when each variable has a single definition [5]. To do this, we create a single-static-assignment form of a program.

A single-static-assignment (SSA) form of a program is an intermediate representation in which each variable has only one definition. Figure 2.3 shows SSA form of the program displayed in Figure 2.2. As shown in the Figure 2.3, each variable has only one definition. This enables us to easily trace the use site of each variable. In the figure, we can also see that two control-flow paths merge into a block. The execution of each control-flow path depends on the initial value, so we can't determine which variable carried the value to the block. In this case, we need a function that will assign a variable based on the control-flow path used to reach the block. We call this function a phi function. Phi-functions are positions in the SSA form of a program where two or more control-flow paths merge. In creating the SSA form, every variable will be set to have one definition, and phi-functions will be inserted in a position where two or more control-flow paths merge.

Transformation of a program to SSA starts with inserting phi-functions for each variable in control-flow graph nodes with two or more incoming control-flow edges. Then, we rename each variable V in a control-flow graph by adding subscript i (for $i = 1, 2, 3, \ldots, n$), where n is the number times variable V is defined in the

control-flow graph, including assignments for phi-functions.

The above approach creates unnecessary phi-functions, which conceals information during optimization and analysis, so we need to create *minimal SSA*. A program is said to be in minimal SSA form if the number of phi-functions is as small as possible. To do this, we discuss an approach introduced by Cytron *et al.* [13].

To use this approach first, we compute the *dominance frontiers* of every node in a control-flow graph. We say node x *dominates* node y if node x appears in every path that leads to node y. We say node x *strictly dominates* node y if node x dominates node y and node x is different from node y. Dominance frontiers of node x is set of nodes s where node x dominates predecessors (nodes that appear after node x and before elements of node s) of nodes in set s but does not strictly dominate elements of set s. After computing dominance frontiers for each node, we insert a phi-function in node y for a variable v in node x, if node y is in the dominance frontier node x.

In this work, we used breadth-first search to create a linearized representation of a control-flow graph from the SSA form of a program. SSA made it easier for us to extract contextually-related blocks of instructions by referring to control-flow jumps and phi functions. We explain the procedure used to transform SSA to a linearized control-flow graph in the next chapter in Section 3.1 (input representation).

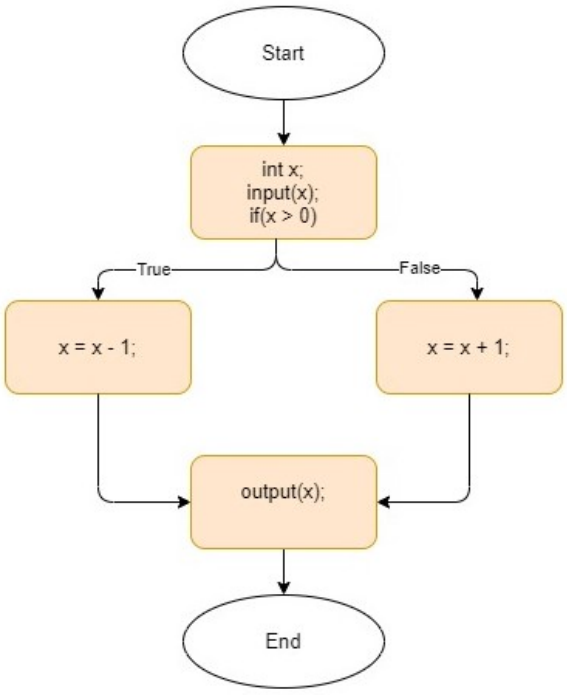

Figure 2.2: Control-flow graph

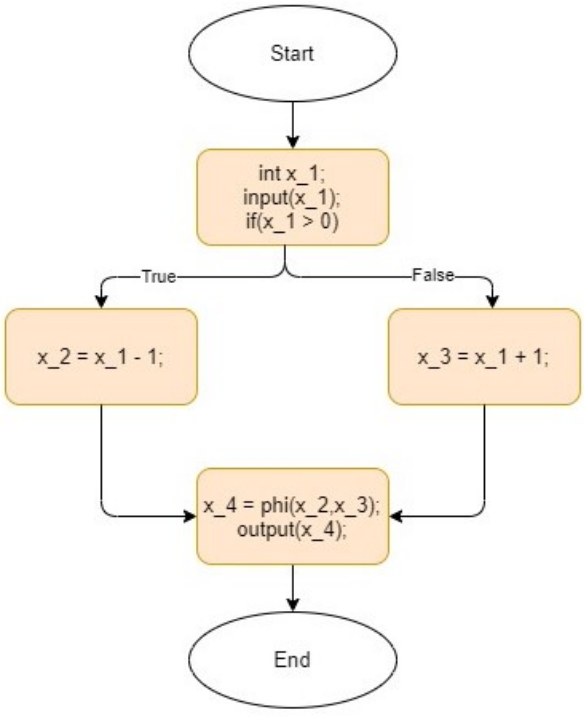

Figure 2.3: Single static assignment form

2.5 Summary

Source-code summarization systems aim to transform source-code artifacts into a natural-language summary. Source-code summarization systems use text-retrieval-based-, heuristic-based- or data-driven-source-code summarization approaches to generate natural language summary for source-code.

We used a data-driven approach that uses control-flow graphs as an additional input representation in this work. Control-flow graphs connect blocks of instructions based on control-flow dependence. We encoded a linearized version of control-flow graphs to sequence models that transform an input sequence to the desired output sequence.

There are different sequence-to-sequence model architectures. Our study used an architecture designed to learn long-term dependency within the input sequence named LSTM. We used a bi-directional LSTM architecture, which goes through the input sequence from start to end (forward pass) and end to start (backward pass). Also, we applied an attention mechanism that helps the decoder focus on a specific encoder state while generating an output sequence. We evaluated our approach using the BLEU score and expert evaluation. The BLEU score measures the n-gram precision of the predicted output compared to a reference text.

3 Experimental Design

In this chapter, we describe our experimental methodology in reference to other related works. We discuss design decisions taken to complete the project in four sections:

- Input representation (Section 3.1): discusses representation used by related works and presents the representation we used to train our model.
- Model architecture (Section 3.2): presents model architecture used by other researchers and explains the model architecture used to accommodate our representation.
- Dataset description (Section 3.3): explains the data collection and cleaning procedures applied to create the final dataset used to train our model.
- Evaluation (Section 3.4): explains the evaluation procedures and metrics used to compare and contrast to related work.

Figure 3.1 illustrates the input and output of the core components of the project. As shown in the figure, we give a raw dataset containing parsed Javadoc comments of a method, SSA form of a method, and the method itself to the data processor. Then, we train the model using the output of the data processor component. Finally, we evaluate the trained model, which predicts a summary given a tokenized control-flow graph and tokenized method.

3.1 Input Representation

This section discusses data representations explored by researchers for data-driven source-code summarization. We extend our discussion from Section 2.4 to assist readers to clearly see the difference between existing approaches and our selected representation. LeClair *et al.* [40] divided work that come under data-driven source-code summarization into two based on the use of AST as an additional input. The first category includes works that used a sequence of source-code tokens [32, 46] and API sequences [29, 47] to represent source code. API sequences are created by extracting and sequencing API calls inside a method. The second category [4, 19, 28, 41, 43] used AST as an additional input beside source-code as a sequence of tokens. Table 3.1 provides a summary of model performances in BLEU evaluated on a dataset by LeClair *et al.* The results shown in the table are collected from evaluations conducted by LeClair *et al.*

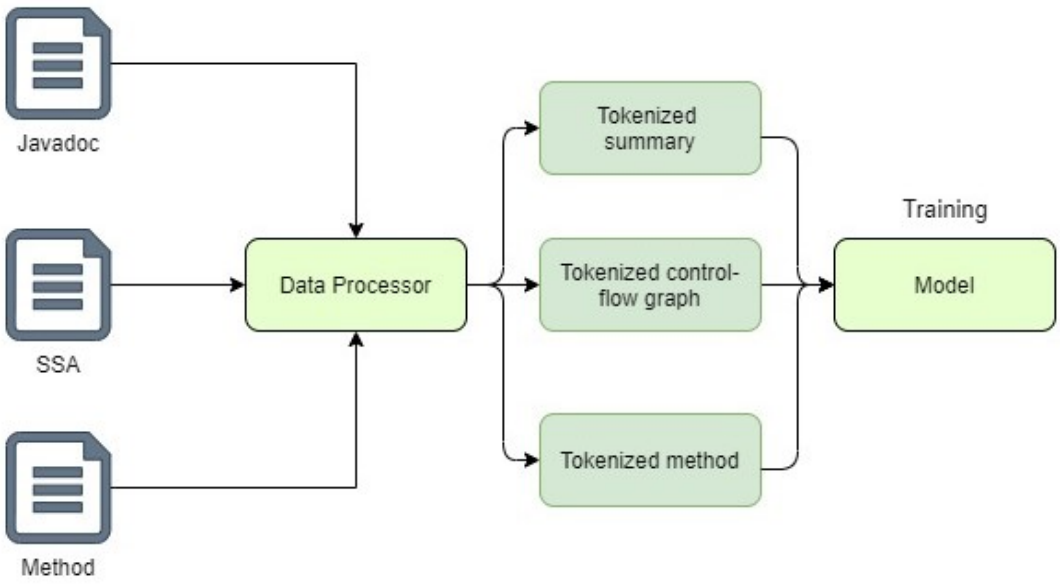

Figure 3.1: Project Workflow

Work that used a sequence of source-code tokens as a sole input explored the application of different types of sequence-to-sequence modeling architectures in a source-code summarization setting. We will discuss these architectures in Section 3.2. The use of a sequence of source-code tokens to encode program properties was motivated by the work of Hindle *et al.* [26]. In their work, they showed that it is possible to model source-code using statistical language models. This motivated work that use data-driven approaches to encode source-code properties into machine learning models.

Also, researchers including Lu *et al.* [47] explored the use of API-sequences for source-code summarization. They proposed a summarization model that outputs comments given API-sequences. Hu *et al.* [29] did a similar study, but their model takes both API-sequences and a sequence of source-code tokens as an input. Both works used a different dataset, but Hu *et al.* compared their model against two baseline models: a model that takes only a sequence of source-code tokens and a model that takes only API sequences. The reported result shows that encoding a sequence of source-code tokens and API sequence is better in precision and recall than a model that takes either a sequence of source-code tokens or API-sequences as an input.

Two works extracted a representation from parse trees. First, Fernandes *et al.* [19] used combination of a sequence model and a graph neural-network to encode source code into a summarization model. A graph neural network aggregates and outputs information encoded in a graph by learning the relationship between nodes of graphs [71]. They constructed the graph by connecting tokens separated based on underscores. Also, they connected subsequent tokens. In addition, they joined tokens that have a data-flow relationship and

tokens connected on a parse tree. Second, Liang and Zhu [43] designed a source-code summarization model that takes parse tree as an input. Both works used Rouge-n [44] to evaluate the generated summaries by the models. Rouge-n measures n-gram recall between the predicted summary and a set of reference summaries. The result of Fernandes *et al.* the addition of structural information using graph neural networks increased the Rouge-2 (measures bi-gram recall) score from 15.3 to 20.8. The model of Lhu and Zhu also outperformed the sequence-to-sequence model that only takes method-token sequences in seven of nine projects used to evaluate the models. The result of both works show addition of structural information of programs help better capture program properties in machine learning models.

Also, the use of intermediate representations, particularly AST, to encode syntactic relation with program tokens along with a sequence of tokens was explored by researchers. Under this category, we reviewed two types of AST representations used by researchers. First, the use of linearized AST for source-code summarization was studied by Hu *et al.* [28] and LeClair *et al* [41]. Second, Alon *et al.* [4] studied the benefit of encoding AST paths between randomly selected terminal nodes for source-code summarization. Both approaches have been shown to be effective and got superlative results with a C# dataset published by Iyer *et al.* [32] and Java dataset by LeClair *et al.* [42]. Based on a comparison conducted by LeClair *et al.* [40], encoding AST into a graph neural-network along with a sequence of source-code tokens outperformed other encoding schemes on the task of source-code summarization. BLEU score was used to compare results. We show the results in Table 3.1.

The related work discussed above show positive results regarding encoding structural information in source-code summarization models. Further, we would like to extend this notion with a lower-level and more detailed intermediate representation that captures relations between blocks of program instructions based on control-flow dependence.

Table 3.1: Model performance in BLEU on a dataset by LeClair *et al.*

Model (Author)	**BLEU score**
ast-attendgru (LeClair *et al.* [41])	18.69
code2seq (Alon *et al.* [4])	18.84
codeGNN (LeClair *et al.* [40])	19.93
codeNN (Iyer *et al.* [32])	9.95
SBT (Hu *et al.* [28])	14.00

In our work, we used the SSA form of a program to create a linearized control-flow graph to represent source code along with a sequence of source-code tokens. Ben-Nun *et al.* [7] explored the use of control-flow graph to encode program properties to neural models. They introduced a term called *contextual-call graph*, a

graph built by connecting nodes based on control-flow relationship. They adopted the notion of semantic similarity based on the contextual relationship from the *distributional hypothesis* [24]. The distributional hypothesis states that *words that occur in the same context tend to have similar meanings*. To apply this idea for programming languages, they defined context as statements whose execution paths depend on each other. Then they stated, *statements that occur in the same context tend to have similar meanings*.

```
public String even_or_odd(int) {
    EvenOdd this;
    int num, temp$0;
    String result, temp$1, temp$2, result_1, result_2,
                   result_3;
    result = "";
    temp$0 = num % 2;
    if temp$0 = = 0 goto label1;
    goto label2;
    label1:
        temp$1 = "even";
        result_2 = temp$1;
        goto label3;
    label2:
        temp$2 = "odd";
        result_1 = temp$2;
    label3:
        result_3 = Phi(result_2, result_1);
        return result_3;
    }
```

Listing 3.1: SSA form of even or odd method

We adopted Ben-Nun *et al.'s* [7] distributional hypothesis for programming language statements to encode program properties for source-code summarization. We used breadth-first search to transform the SSA form of a program method to create a linearized control-flow graph. After setting the context distance to one, we created the linearized control-flow graph by connecting any two blocks on the control-flow graph based on the following conditions: First, blocks with a direct control-flow relationship are connected by referring to go-to statements and phi-functions. Second, if a block doesn't have a control-flow parent node, we connected it to the previous node unless it is the root node, and in case the previous block only contains a single go-to statement, the block with no control-flow parent will be connected to the parent of the previous block. Listing 3.1, Figure 3.2, Figure 3.3, and Figure 3.4 show the input, intermediate representations, and output of our data representation approach. The first step is converting the source-code shown in Figure 3.2 to the SSA form shown in listing 3.1. Then, we create a

linearized version of the control-flow graph shown in Figure 3.3 by creating a block sequence shown in Figure 3.4.

Overall, source-code summarization models benefit from encoding the structural information of a program. We extended this idea with a representation that shows contextual relations between program instructions based on execution flow. Following this, we constructed a linearized version of the control-flow graph that shows contextual relation between blocks of instruction based on contro-flow dependence. We turned this representation into a sequence and used it as an input in a source-code summarization model.

```java
public String even_or_odd(int num){
    String result = "";
    if(num%2 == 0){
        result = "even";
    }
    else{
        result = "odd";
    }
    return result;
}
```

Figure 3.2: Example Java method that determines if a number is even or odd

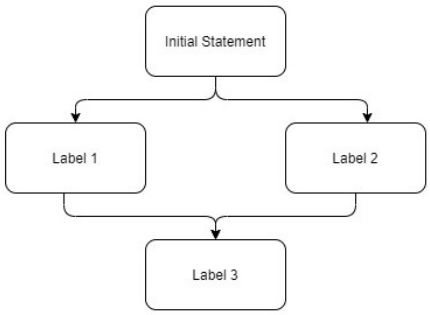

Figure 3.3: Control-flow graph of the even or odd method

```
['initial_stmt',
 'label1:',
 'initial_stmt',
 'label2:',
 'label1:',
 'label3:',
 'label2:',
 'label3:']
```

Figure 3.4: Block sequence generated for the even or method from the SSA form

3.2 Model Architecture

This section discusses model architectures proposed and evaluated by researchers to summarize source code. Like Section 3.1, we extend this discussion from Section 2.2 to clearly identify existing approaches and the selected model architecture. Source-code summarization is a machine translation task, similar to translating a text from one natural language to another (e.g. English to French). For the task of automated source-code summarization, researchers used neural networks such as GRU [12] and LSTM [27] with Bahdanau- [6] and Luong-attention [48]. All models discussed in this section were evaluated using the BLEU score [55], and some works used METOR score [15] and Rouge-N [44]. METOR score like ROUGE-N is a recall-oriented

score that measures uni-gram overlap between the predicted and the output sequence.

Iyer *et al.* [32] and Lu *et al.* [47] used LSTM based sequence to sequence model for source-code summarization. First, Iyer *et al.* trained an attention-based LSTM model named Code-NN to summarize C# and SQL code. Their model takes a sequence of tokens as an input. Second, Lu *et al.* proposed and evaluated an attention-based LSTM model that takes API sequences to generate a summary. The difference between the two models is that Iyer *et al.* used the Luong-attention, and Lu *et al.* used Bahdanau-attention. Iyer *et al.* compared their approach with an information-retrieval-based summarization system, statistical machine translation system, and a sequence-to-sequence model with no attention mechanism. They used both intrinsic evaluation and extrinsic evaluation. They used BLEU and METEOR scores for the intrinsic evaluation. The extrinsic evaluation was an expert evaluation where the participants rated the generated summaries for naturalness and informativeness. The naturalness measures grammatical fluency, and the informativeness measures the amount of information captured by the summary regardless of fluency. Their model got superlative results in both evaluations. Lu *et al.* evaluated their approach using the BLEU score. They evaluated their model in different settings, and their result shows the attention-based sequence-to-sequence model is better than a sequence-to-sequence model with no attention.

Hu *et al.* [28] trained an LSTM model for summarizing source-code. Alon *et al.* [4] used a GRU sequence-to-sequence model to summarize methods. Hu *et al.* encoded a linearized AST to a GRU model that uses Bahdanau-attention. Alon *et al.* also used a GRU model with Luong-attention to encode randomly selected AST paths between two terminal nodes and generate a summary for the method. On evaluation conducted by Hu *et al.*, their model outperformed the model proposed by Iyer *et al.* On comparison made by Alon *et al*, the model proposed by Alon *et al.* outperformed models proposed by Iyer *et al.* and Hu *et al.* Both evaluations used BLEU score. We can also see the performance difference between the models in Table 3.1, where models by Alon *et al.*, Hu *et al.*, and Iyer *et al.* got 18.84, 14.00, and 9.95, respectively.

Hu *et al.* [29] and LeClair *et al.* [41] trained a multiple encoder GRU model for source code summarization. Hu *et al.* trained a model that encodes API sequences in addition to a sequence of source-code tokens. They used Bahdanau-attention on the output of each encoder to create a context vector. Then, they took the sum of the context vectors to create one context vector used by the decoder. LeClair *et al.* proposed and evaluated a model that takes tokenized AST beside a sequence of source-code tokens. They used Luong-attention. They concatenated the context vector from both encoders and fed it to the decoder to generate a summary. LeClair *et al.* compared their model with Hu *et al.'s* model [28], and the reported result shows that LeClair *et al.'s* model outperformed the model proposed by Hu *et al.* The evaluation metric used for comparison was BLEU score. The model by Hu *et al.* got 14.00, and the model by LeClair *et al.* got 19.6.

Also, LeClair *et al.* [40] proposed and evaluated a model that used a graph neural network to encode AST along with a GRU encoder for a sequence of source-code tokens. This model is the current state of the art in source code summarization. This work also reported that the model by Alon *et al.* [4] outperformed LeClair *et al.'s* model [41]. The graph neural network-based model outperforms both models. The result of the evaluation is shown on Table 3.1.

In this work, we adopted architecture proposed by LeClair *et al.* [41] with small modifications. The model is designed to encode multiple input representations. First, we changed the RNN cell type from LSTM to GRU. Britz *et al.* [11] showed LSTM consistently outperforms GRU in neural machine translation tasks. Second, we used a bi-directional encoder instead of a uni-directional encoder after referring to the result published by Britz *et al.*, which showed that bi-directional encoders yield a better result in terms of precision than uni-directional encoders in neural-machine translation tasks. Also, we considered that the control-flow graph encoder would be benefited by looking at the given sequence from beginning to the end and end to the beginning of the sequence since we set a maximum sequence length of 289 for the control-flow graph representation. Also, we have a second model that only takes a sequence of method tokens. The architecture is similar to method_cfg model, but it does not have the control-flow graph encoder and the subsequent attention mechanism to align the control-flow graph input to the output of the decoder.

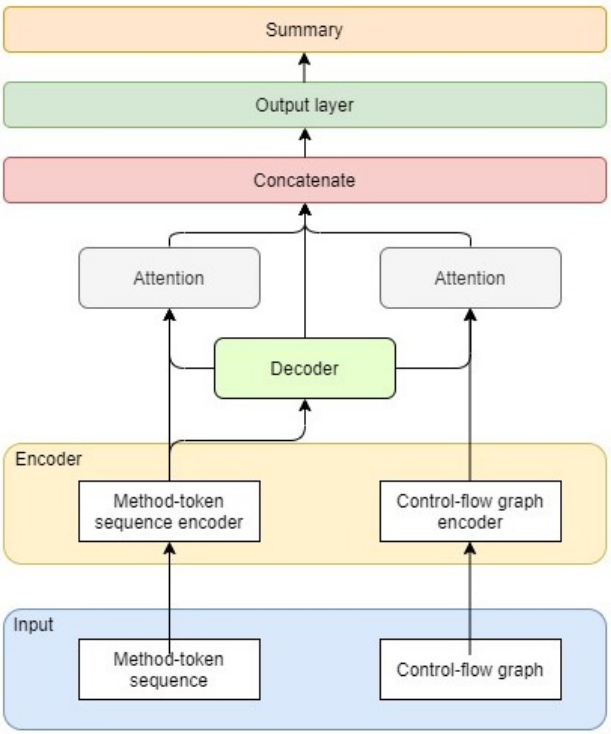

Figure 3.5: Architecture of method_cfg model

Figure 3.5 shows the structure of the model used to demonstrate our approach. As shown in the figure, the model has two encoders and one decoder. On top of the encoders, there are two attention layers that apply a global attention mechanism by Luong *et al.* on the output of each encoder. The model concatenates outputs from the two attention layers and the decoder on the merge layer. Finally, the concatenated matrix through a dense layer and the output of the dense layer will be passed to the final softmax layer. Further modifications were also made while tuning the model. We will discuss them the implementation chapter.

Overall, multiple architectures with different input representations have been shown to be effective for source-code summarization. The evaluation results show source-code summarization models benefit from the inclusion of structural information besides method-token sequence. To encode the structural information, we adapted a multi-encoder architecture proven to be effective in encoding AST beside the method-token sequence.

3.3 Dataset Description

This section presents the source of our dataset and the approaches used to process the dataset. For this study, we decided to create our own dataset since we had to compile projects to extract the control-flow graph from the SSA form of the program.

There are two ways to split a source-code summarization into training, testing, and validation sets. First, we can split a dataset at the function (method) level. This type of split divides the dataset using method-summary pair as a reference point. In a dataset split using this mechanism, we can find method-summary pairs in both the training and testing sets of the dataset. A study done by LeClair *et al.* [42] shows this type of split inflates the evaluation score of a model under study. Second, we can split a dataset at the project level. In this type of split, method-summary pairs from a project will not end up in both the training and testing sets of the dataset. This type of split creates a scenario that the model under study gets to be evaluated by the project it has not seen before.

Our review to conduct this study found two existing method-level datasets for data-driven source-code summarization for the Java programming language. First, Hu *et al.* [29] created a source-code summarization dataset from Java projects on Github [21] with a minimum of 20 stars created between 2015 and 2016. The dataset contains 69,708 data points and has been used by other related work [2, 69]. Hu *et al.* did not mention project-level dataset splits when creating their training and testing sets. This indicated a function-level splitting strategy was used for dividing their dataset into training and testing sets for their study.

Second, LeClair *et al.* [42] published a source-code summarization dataset with 2.1 million data

points (method-summary pairs). The dataset is extracted from the Sourcerer project published by Lopes *et al.* [45]. The published dataset is already split into training, validation, and testing sets at the project level and released to promote reproducibility among models that take methods as a sequence of tokens and output a summary of the input.

Our work involved an additional input, which is a control-flow graph extracted from the SSA form of a program along side the method as a sequence of tokens. To generate this input, we were required to collect compilable projects with all required dependencies made available and pass it to the framework used to compute the SSA form of the program. This led us to collect compilable projects and create a new dataset with a control-flow graph as an additional field. In creating the dataset, we adopted guidelines forwarded by LeClair *et al.* We discuss the guidelines as we go through our data processing steps in Section 3.3.2.

In the following subsections, we discuss the dataset collection method and data processing steps taken to create the final dataset used to train our models.

3.3.1 Dataset Collection

For this study, we started with a dataset provided by Martins *et al* [49]. They published 50,000 compilable Java projects available on Github. We used three components of the dataset: an archive of compiled projects as bytecode, an archive of Jar files required to build the projects, and a text file containing a list of Github URLs for compilable projects.

We cloned the projects from Github by reading the project URL listed on the text file. Next, we parsed the project files to create our raw dataset, which maps methods to their leading preceding statements.

Out of the 50,000 projects, we were able to clone 42,640 projects. The rest of the projects could not be found with the given project URL. From this, 22,780 projects contained at least a method with a preceding Javadoc statement. The raw dataset constructed from these projects collectively contains 790,580 data points, and on average, there are 34.7 documented methods per project.

Next, we explain the steps taken to process the raw dataset to create the final dataset.

3.3.2 Data Processing

After creating the raw dataset, we filtered and matched the comment-method pairs in the raw dataset to the control-flow graph we created from the SSA form of the projects. We used the steps enumerated below to clean up the raw dataset.

Step 1: Cleaning Javadoc Statements and removing noise from the dataset

1. We split Javadoc statements using a period as a delimiter and took the first sentence as a summary for methods.

2. We used a library named langdetect [38] to filter out non-English summaries.

3. We filtered out comments that do not provide information about the method by indexing Javadoc tags (e.g., `@author`,`@version`, and `@since`).

4. We removed unit-test methods with a summary that gives a vague description of the method (e.g., the phrase *"test case for"* followed by the name of the method and similar summaries such as *"create the test case"*). We applied a heuristic that removed data points with a comment containing the *"test"* token. This step removed 3.01% of the dataset.

5. We filtered out empty methods with leading Javadoc statements. This step removed 6.38% of the dataset.

Steps 1 and 2 of this cleaning step are adopted from LeClair *et al.* [41]. Collectively, the processing steps described above removed 38.4% of the comment-method pairs and reduced the dataset size to 487,036.

Step 2: Cleaning Java methods

1. We removed block- and inline-comments from the methods. Then, we replaced constants with their type name identifiers (e.g., num and str).

2. We removed punctuation from the methods and filtered out empty methods created when we remove punctuation.

3. We converted the methods to AST in XML format. This representation was used as an input for the data processor of codeGNN model.

This processing step removed 7.2% of the filtered dataset from the previous step, and the size of the dataset became 451,962. At this point, we filtered out 33.06% of the projects in the raw dataset, which resulted in 15,250 unique projects.

Step 3: Creating and cleaning control-flow graphs

1. We copied the bytecode of the projects in the dataset to a directory from the archive of built projects released with the dataset. We managed to transform 59.89% of the remaining projects into SSA form, and we ended up with 9,133 projects in the dataset. We lost 40.11% of the projects due to compilation errors that occurred while transforming Java bytecode to SSA.

2. We kept the data points with corresponding SSA form by filtering based on class and project name, and this removed 47.6% of the filtered dataset created by the previous filtering step, and 236,827 data points remained in the dataset.

3. We constructed, parsed, and cleaned the control-flow graph from the SSA form of projects.

4. We matched control-flow graphs to their corresponding method-comment pairs in the dataset based on method- and class-name. This step removed 42.34% of the filtered dataset and reduced the dataset size to 136,547 data points with 7172 unique projects.

Step 4: Final Cleaning Steps

1. We performed a final filtering step based on summary- and control-flow graph sequence length. We used interquartile range (IQR) [30] outlier detection to get the upper bound of the sequence length for the summary and control-flow graph. Equations 2.1 to 2.3 show the formulas to get upper and lower bound of a distribution. Using this approach, we set the upper bound of the summary sequence length to 16. We set the lower bound of summary sequence-length to 3 following LeClair *et al.* [41] . For the control-flow graph, we removed data points with control-flow graph sequence-length greater than 289. We also set the maximum AST node-list length to 100 following LeClair *et al.* [40]. The list of AST nodes is one of the inputs for the state-of-the-art model. Setting the maximum AST node-list length to 100 made the maximum method sequence length to 58.

2. We removed data points with the same method, summary, and control-flow graph. This filtering step removed 28.89% of the filtered dataset from the previous step, and the dataset size was reduced to 97,917 data points.

3. Finally, we tokenized each input and make it ready to be used to train the models.

Table 3.2 provides a statistical summary of our final dataset. It presents information based on each representation's sequence length and the number of tokens in each representation. The sequence length column each representation's average sequence length, median sequence length, and percentage of data points below the average sequence length. In the number of tokens column, we presented the number of tokens and the number of unique tokens for each representation.

As shown in the Table 3.2, 65.65% of the final dataset has a method-tokens sequence length less than 13.13. The mean sequence-length for this representation, 13.13 by itself, is small. We checked if this happened when we filtered and matched summary-method pairs to their corresponding control-flow graph. We found that the mean and median after cleaning the summary and method are 27.67 and 15.0, respectively. The percentage of data points below the mean is 74.56%. Figure 3.7 also shows the final dataset distribution for method-tokens sequence length, and we can see that the distribution

$$IQR = Q_3 - Q_1 \tag{3.1}$$

$$lower\_bound = Q_1 - 1.5IQR \tag{3.2}$$

$$upper\_bound = Q_3 + 1.5IQR \tag{3.3}$$

Outlier Detection

Table 3.2: Dataset Statistics

	Sequence Length				**Number of Tokens**	
	Max	**Mean**	**Median**	**<Mean**	**All Tokens**	**Unique Tokens**
Method	58	13.13	11.00	65.65%	1,285,225	12,674
Control-flow graph	289	67.08	52.00	67.32%	6,568,509	24,174
Summary	16	9.38	9.00	68.3%	918,309	26,904

has a heavy tail to the right. When we come to the control-flow graph representation, Figure 3.6 shows the sequence-length distribution has a long right tail and is highly skewed to the left. The mean and median sequence length of this representation is 67.08 and 52.00, respectively. Figure 3.8 shows the distribution of the final dataset with respect to summary length. We can see that it is a normal distribution.

In Table 3.2, we can also see the number of unique tokens (vocabulary) for each representation. Compared to LeClair *et al.* [42] and Hu *et al.* [29], both the input and output vocabulary are relatively small. Because of this, we did not use an out-of-vocabulary token to replace tokens with a low-frequency value for each representation in the dataset.

Overall, we have created a source-code summarization dataset that maps method-token sequences with summary, control-flow graph, and AST. We started with cleaning method and summaries. Then, we compiled projects to extract control-flow graphs. Finally, we matched the control-flow graphs with method-summary pairs and tokenized the dataset.

3.4 Evaluation

We evaluated our approach with two experiments. For the first experiment, we use BLEU score to compare three models. In the second experiment, we employ human judges to rate candidate translation from three models.

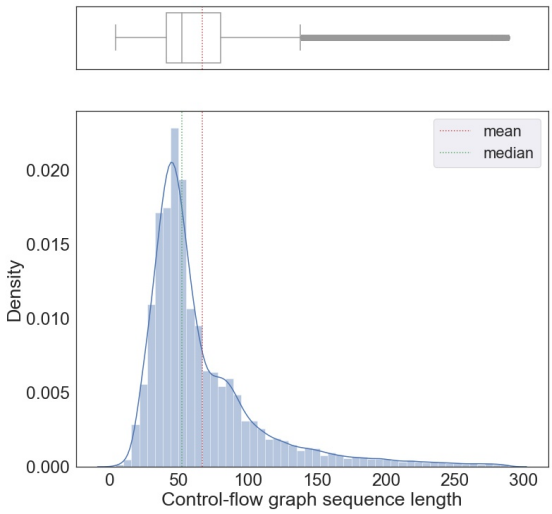

Figure 3.6: Control-flow graph sequence length distribution

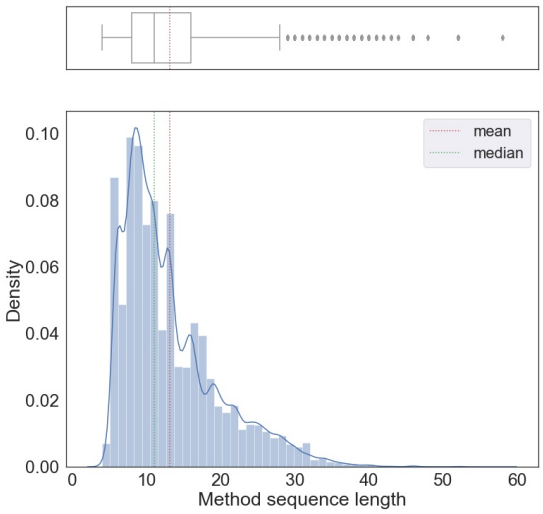

Figure 3.7: Method-token sequence length distribution

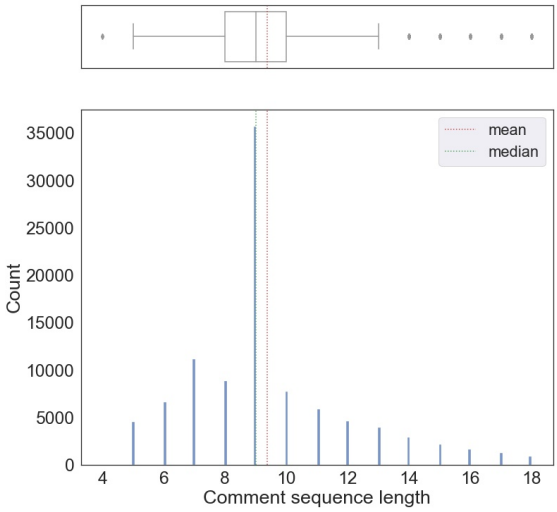

Figure 3.8: Comment sequence length distribution

3.4.1 Automatic Evaluation

In this experiment, we evaluate the performance improvement achieved by encoding an control-flow graph alongside a sequence of source-code tokens for summarization models according to n-gram precision. We call the proposed model *method_ cfg*.

For this experiment, we compare the method_cfg model with two other models. We implemented the first model that encodes a method as a sequence of tokens to output a summary. We call this model *method_ only*. The architecture of the model is a single input encoder-decoder architecture with a global-attention mechanism. The model parameters are also the same as the sequence of source-code tokens encoder and decoder of the proposed models. The second model is the state-of-the-art model by LeClair *et al.* [40]. We call this model *codeGNN*. We used code+gnn+BiLSTM model, which is the top performing model according to results showed by LeClair *et al.*

In addition, we trained and evaluated the *method_ only* on a dataset by LeClair *et al.* [42] to compare the model with other works.

3.4.2 Expert Evaluation

In this experiment, we conducted an expert evaluation to compare summaries generated by method_cfg, method_only, and codeGNN. We employed 26 computer science students: 19 undergraduate students, 3 MSc students and 4 Ph.D. students. Participants rated summaries generated by the candidate models for 20 methods with respect to describing the functionality of the sampled methods correctly. We used a 5-point Likert scale to rate each summary, with one being the lowest and five being the highest.

3.4.3 Data Sampling

We randomly sampled 20 methods and their corresponding summaries generated by each model on the testing set. We accounted for two criteria for sampling the candidate method-summary pairs. First, we filtered out short methods (such as getters and setters) that are easy to predict by the candidate models. Second, the number of method-summary pairs in the dataset contributed by each project differs significantly. Many projects contributed a small number of method-summary pairs (1 to 10 method-summary pairs per project), while few projects contributed a large number of projects. We can see this by the log-log project-frequency distribution in Figure 3.9. Considering this, we selected candidate method-summary pairs both from projects with a small and large contribution to the dataset. This enabled us to avoid the candidate methods and their corresponding summaries being sampled from a small group of projects.

We first use the method sequence-length frequency distribution shown in Figure 3.7 to filter out

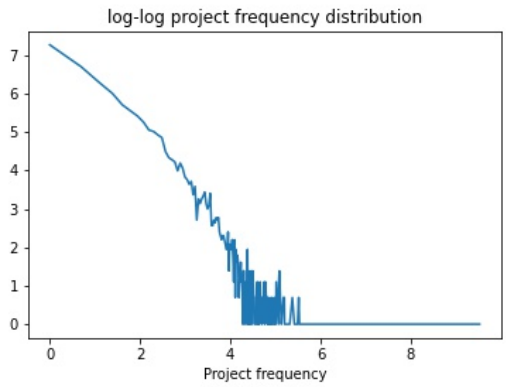

Figure 3.9: Log-log project-frequency distribution

short methods. We applied this filtering step twice. First, we removed data points with method sequence-length below the median sequence length. For this filter, the median is 11.0. Second, we removed short methods below the median length from the output of the first filter. For this filter, the median is 16.0. Then, we used the project-frequency distribution to group projects sorted based on frequency into four categories. With this, we grouped the projects into four based on their contribution to the dataset. From each group, we sampled five methods and their corresponding summaries generated by each model.

All in all, we sampled the candidate methods and the predicted summaries by each model by filtering based on method sequence-length frequency and grouping based on project frequency.

3.4.4 User Interface

The expert evaluation form can be accessed by clicking this link. The first page is an introduction page. It explains the content of the form and prompts users to submit their educational background, general programming experience, and Java programming experience. For educational backgrounds, we asked the user to choose one of the following categories: undergraduate student, MSc student, Ph.D. candidate, faculty, and other specified by the participant. We use a text field as a prompt for programming experience and Java programming experience.

The remaining pages contain the evaluation form for each method. A page contains a method, three summaries generated by the candidate models for the method. The users were asked to rate each summary with respect to correctly describing the functionality of the method. We used a radio button to collect ratings for each summary on a scale of 1 to 5. The qualitative descriptors of the scale are 1 — strongly disagree, 2 — disagree, 3 — neutral, 4 — agree, and 5 — strongly agree. At the end of the page, there is an optional text field to submit their own summary.

3.4.5 Model Selection

We trained all models for 20 iterations (epochs) and saved the model weight at each epoch. It took 1 hour and 50 minutes to train the method_cfg model. For method_only model, it took 55 minutes to train the model. The number of epochs is one of the training parameters we selected to train all models. We selected method_cfg and method_only model by first evaluating their performance measured by BLEU scores on the test set. For this evaluation, the method_cfg predicted the summaries given encoder inputs (method-token sequence and control-flow graph) for and decoder inputs (summaries). Also, the method_only model predicted summaries given method-token sequence and summaries. The result of this evaluation is not the final evaluation result because the models were given the summaries beside the encoder inputs, but it can be used as an intermediate result that shows the performance of the model on the test set. This is similar to validation, but we used BLEU score performance on the test set rather than using accuracy or loss over validation set during training. The reason behind this is the validation during training does not measure the n-gram precision, rather it is the average of prediction accuracy of the model on each output timestep. For both method_cfg and method_only models, we selected the model that scored the highest BLEU score among the models saved at each epoch. We then used an inference model to predict the summaries given the encoder inputs. The inference model predicts the output summary given only encoder inputs.

We also trained the codeGNN model for 20 epochs. It took 98 minutes to train the model. Then, we adopted the model selection approach LeClair *et al.* [40] used in their study. They selected the model that scored the highest validation accuracy. During training their model, six models scored the highest validation accuracy. Then, we used their inference model to predict summaries given method, and AST nodes and edges. We then measured the BLEU score of each model and selected the prediction of the model that scored the highest BLEU score.

3.5 Summary

In this chapter, we covered the data collection and cleaning procedure and the architecture of our model, including the input representation. Related works showed that source-code summarization models benefit from encoding the structural information of a program. With this, we chose a representation that shows the flow of execution and captures the control-flow relation between blocks of instruction. To use this new representation, we chose a multi-encoder architecture that concatenates the encoded information and uses it to predict a code summary. To evaluate our work, we used both an automatic and expert evaluation.

4 Implementation

This chapter describes implementation details of components used in the research project. We presented the description in four sections. The first three sections, dataset collection and processing (Section 4.1), linearized control-flow graph (Section 4.2), and model implementation (Section 4.3), discuss the approach used to implement components of the project. In training details (Section 4.4), we describe model configurations used to train the proposed model and baseline models.

4.1 Dataset Collection and Processing

This section discusses the approach used to collect, clean, and filter the raw data to construct the training dataset.

We implemented a Java program to parse project files and collect methods with leading Javadoc comments from projects we cloned from dataset provided by Martins *et al.*[50]. This program uses Eclipse's AST parser version 3.7.1 to extract method and Javadoc pair from streamed Java files. This filters out the methods without leading Javadoc comments and abstract methods. This program's output is our raw dataset used to create the final dataset used to train our models.

After generating the raw dataset, we started filtering the dataset based on conditions set to collect cleaned summary. The following filter conditions are applied to prepare the final summary used to train our models.

- We split the Javadoc statement using a period as a delimiter and selected the first element of the list. If a Javadoc statement does not have a period, we used "@param" and "return" tags to split the statement.
- We removed Javadoc formatting tags such as "@link" and "<code>" from the summary and kept the remaining comment tokens.
- We used a library called langdetect 1.0.8 to filter out non-English comments.
- Finally, we removed punctuation and digit from the remaining summaries.

After we prepared the summary, we filtered the dataset based on conditions set to clean method tokens.

- We removed inline and block comments from the method.

- We replaced constants with type identifiers “str” for Strings, “num” for integers, and floating-point numbers.
- We filtered methods that do not have a body and methods that became empty when we remove comments.
- Finally, we removed punctuation from the remaining methods.

The next step of filtering involved removing method-comment pairs that do not have corresponding control-flow graphs. We will discuss the approach used to create the control-flow graph in the following section.

Finally, we tokenized each input and applied a final filter based on sequence length, as discussed in Section 3.3.2. The output of this step is the dataset used to train and test the models.

4.2 Linearized Control-Flow Graph

In this section, we explain the approach used to transform a Java program to SSA form. Then, we discuss the method to create a linearized control-flow graph from the SSA form of a program. Finally, we list data cleaning steps applied to the generated control-flow graphs.

To transform the Java programs into SSA form, we used a compiler framework called Soot 4.2.1 [68]. This framework transforms a Java program to a representation called *Shimple*, an SSA form of another representation that can be generated by the framework called *Jimple*. Jimple is a three-address representation of Java bytecode [68].

As we discussed in Section 3.3.1, the dataset used to create the training dataset has three components an archive of compiled projects as bytecode, an archive of Jar files required to build the projects, and a text file containing a list of Github URLs for compilable projects. To convert the bytecode of projects with method-comment pairs in the filtered dataset to SSA, we first parsed the JSON file we found in each project directory that contains the build information of projects to categorize the projects into two groups: projects that do not require additional Jar files to compile and projects that require additional Jar files to compile. Next, we copied the bytecode of each project into a directory in the project folder. For the projects that do not have a dependency, we copied the jar format of the `soot` framework into the project folder. Then, we passed the directory that contains the project bytecode as an argument for the soot command-line application. For the projects that required Jar files, we wrote an additional script that copies and extracts the required Jar files in the project directory. Then, we used the same script we used for the projects that do not require additional jar files, which copies the soot jar file and passes the project directory as an argument for the soot command-line application. The whole process was done automatically. Even though the program helped us

speed up the data generation process, the program could not transform 40% of the projects in the filtered dataset. This occurred due to the following reasons:

- Some projects required dependencies not available in the Jar-file directory of the dataset.
- A compilation error was raised by the compiler framework while transforming the bytecode of some projects to SSA.

After transforming projects to SSA form, we selected the SSA form of class files of the projects that remained in the filtered dataset and copied them into a directory. Next, we parsed the files to create a data frame that indexes the SSA form of a method using class and method name. Then, we passed the data frame to the program that creates a linearized control-flow graph from the SSA form of a method.

Algorithm 4.1 shows the pseudo-code of the technique we used to transform SSA into a linearized control-flow call graph. The input for the procedure is the SSA form of a method, and it outputs a sequence of contextually related blocks. The first step of the process is creating dictionaries that associate blocks with a list of the block statements. Then, we create two dictionaries that associate the start and destination of control-flow relationships. This relationship is captured with goto statements and phi-functions. Finally, we iterate through a list of blocks from the block dictionary and connect each block based on the control-flow relationship by referring to goto and phi-functions. In our heuristic, we found a special case that the control-flow parent of the "catch" block of an exception cannot be found in phi and goto dictionaries. To handle this issue, we connect such types of blocks to the parent of the previous block or the block before the "try" block.

We now have a linearized control-flow graph constructed from SSA form of a program. Finally, we tokenized the control-flow graph and matched it with method-summary pairs using project name, class name and method name.

4.3 Model Implementation

To evaluate our approach, we implemented a model with TensorFlow Keras 2.4.0 API [37]. We trained the model to output a summary given a sequence of source-code tokens and a control-flow call graph.

The model has two BiLSTM encoders and one LSTM decoder. The architecture of the model is shown in Figure 3.5. One encoder takes a method-tokens sequence, and the second one takes a control-flow graph as an input. During training, the decoder takes the summary as an input. Both the encoders and the decoder have an embedding layer that learns the representation of input tokens during training. After encoding the inputs, we use the global attention mechanism introduced by Luong *et al.* [48]. We compute two context vectors using output from each encoder. We use a dot product between the output of an encoder and the

Algorithm 4.1: Linearized control-flow graph creator

```
Result: path
blockCounter ← 0;
path ← [];
blockDictionary ← createBlockDictionary(SSA);
phiDictionary ← createPhiDictionary(SSA);
gotoDictionary ← createGotoDictionary;
blockList ← blockDictionary.keys();
gotoDestinationList ← createGotoDestinationList(gotoDictionary);
while blockCounter < sizeOf(blockDictionary) do
    block ← blockList[blockCounter];
    if blockCounter >= 2 then
        previousBlock ← blockList[blockCounter − 1];
        secondPreviousBlock ← blockList[blockCounter − 2];
    end
    if block in gotoDictionary.keys() then
        path ← addSequence(path,block,gotoDictionary[block]);
    end
    if block in phiDictionary.keys() then
        temporaryPath ← [block, phiDictionary[block]];
        if checkRepition(temporaryPath) == False then
            path ← addSequence(path,block,phiDictionary[block]);
        end
    end
    if (block not in gotoDestinationList) and (previousBlock in gotoDictionary.keys()) then
        if sizeOf(previousBlock) == 1 then
            path ← addSequence(path,secondPreviousBlock,block);
        end
    end
    increment(blockCounter);
end
```

decoder to compute the alignment score. After computing the context vectors, we adopted a technique by LeClair *et al.* to merge the context vector from each encoder with the output of the decoder. The technique concatenates the context vector from the sequence of source-code tokens encoder, the output of the decoder, and the context vector from the control-flow graph encoder. It passes the resulting matrix to a layer with a tanh activation function. Finally, we give the output of the tanh layer to a layer with a softmax activation function to output one summary token at a time based on the provided information.

4.4 Training Details

For training the model, we used the settings shown in Table 4.1. The model was sensitive to LSTM cell size, embedding dimension, batch size, and the dimension of the tanh layer that comes after merging contexts from both encoders with the output of the decoder.

A rough interpretation of BLEU is outlined in table 2.2. All scores mentioned in this section are average scores where the translation gets the gist but contains grammatical errors. Since BLEU correlates with human evaluation, here we interpret a marginally higher score as a marginally better summary translation.

We explored different configurations for LSTM cell size, embedding dimensions the dimension of the dense layer, and batch size.

First, we found parameters for each encoder by training them separately as an individual model to output summary given a control-flow graph or a sequence of source-code tokens. Both models gave a superior performance with LSTM cell size 256 and with the rest of the parameters configured as the final model parameter shown in Table 4.1. Then, we connected trained both encoders with a single decoder as one model. In this case, setting the LSTM size to 512 improved the performance of the model in BLEU score but the training time increased by 60%. We selected a random fold to show the performance difference between LSTM cell size 256 and 512. On the evaluation the 256 model got a BLEU score of 25.02 while the 512 model got 29.44.

For embedding dimensions, we first trained both encoders with training size 128. The encoder that takes a sequence of source-code tokens has acceptable performance with 128, but the performance improved when we set it to 256. On the other hand, the control-flow graph encoder with embedding size 256 did not show a stable performance across folds of the dataset. On the evaluation set we used above the model we selected got a BLEU score of 29.44, while the model with method-token-encoder embedding size 128 got 26.71.

We trained the model on 256, 512, 768, and 1024 sizes for the dense layer dimension. Out of these parameters, 512 showed a consistent performance across different folds of the dataset. On same evaluation set

the 512 model got 29.44, the 256 model got 29.42, the 768 model got 26.52 and the 1024 model got 29.73. On this evaluation set the 1024 model outperformed the model we selected, but on another evaluation set the 1024 model got 18.63 while our model gets 25.03. We used BLEU score as an evaluation metric.

We observed performance fluctuations during training due to a change in the batch size of the training set. Batch size is the number of training examples the model is seeing at a time to compute an error gradient. A change in training set size caused the performance fluctuations during 10-fold cross validation. We used a batch size of 128 to train the model.

Table 4.1: Model Configuration for Training

Parameter	**Setting**
LSTM cell size	512
Embedding dimension	256
Embedding dimension for Control-flow graph	128
Dense layer for the context vector	512
Dense layer activation	tanh
Loss function	Sparse categorical cross-entropy

5 Evaluation

In this chapter, we present the result of our experiments. Section 5.1 discusses the expert evaluation result. In Section 5.2, we present the automatic evaluation result. Section 5.3 explain the limitations of our approach.

5.1 Expert Evaluation

In this study, we conducted an expert evaluation to compare summaries generated by three models: the proposed model (method_cfg), the model that only takes method token sequence as an input (method_only), and the current state-of-the-art model (codeGNN).

Table 5.2 shows example summaries generated by method_cfg and method_only model. The first example shows method_cfg predicting a summary translation that describes the method but is not similar to the reference translation. In this case, BLEU doesn't consider paraphrase, so the translation won't be given a correct translation score. In the second example, the method_cfg did not predict the correct summary. In this example, the method_only model predicts a similar summary to reference text.

We collected evaluations from 26 computer science students: 19 undergraduate students, 3 MSc students, and 4 Ph.D. candidates. On average, the participants reported 5.29 years of programming experience and 2.48 years of Java programming experience.

Table 5.1: Model performance in BLEU

Model	**BLEU score**
method_only	11.61
method_cfg	9.94
codeGNN	3.21

Table 5.1 shows the performance of models used for the study in BLEU score over the test set used to sample the candidate methods and their corresponding summaries generated by each model. The BLEU score of the candidate models: method_only, method_cfg, and codeGNN is 11.61, 9.94, and 3.21, respectively. As shown in the table, the method_only model outperformed the method_cfg and codeGNN model in terms of

BLEU score over the test set used in the expert evaluation. The method_only model score a BLEU score 16.80% higher than the method_cfg model. Both method_only and method_cfg models outperformed the codeGNN by 8.40 and 6.73 BLEU score points, respectively. Based on table 2.2 that discusses the qualitative interpretation of BLEU scores, the method_only model falls into the second category where it is hard to get the gist of the summary. The method_cfg and codeGNN models fall into the first category where the generated summaries are almost useless.

Table 5.3 shows the statistical summary of the average rating given to the model by the study participants. The table shows the mean average-rating, median average rating, minimum average rating, maximum average rating, and the standard deviation of the average rating. The minimum value for the average rating is zero, which indicates all participants strongly disagree with the given summary for the method. The maximum possible value for the average rating is five, which shows all participants strongly agree with the given summary for the method. Graphically, we show the distribution of the average rating in Figure 5.1. The blue box plot show distribution of the average rating given to codeGNN model. The orange and green box plots show the distribution of the average rating given to method_only and method_cfg, respectively.

Figures 5.1 and 5.2 show that the average rating given to each model is normally distributed. Hence, we used the mean average-rating to compare the performance of each model based on the expert evaluation. Even though the method_only model scored 16.80% higher BLEU score than the method_cfg model, the method_cfg model received 0.58% higher mean average-rating than the method_only model. On the other hand, compared to codeGNN, both method_only and method_cfg models respectively received 1.96 and 1.98 higher mean average-ratings than codeGNN model. The results are mixed: our new technique clearly outperforms the state-of-the-art, but BLEU scores are not reliable in a setting where we only have one reference for each predicted output.

We conducted a statistical significance test on three hypotheses. We used an independent t-test since the result is normally distributed. The statistical significance test tested the following hypotheses:

- $H1_{null}$: there is a significant difference between the average-rating distribution of the method_cfg and method_only models.
- $H2_{null}$: there is a significant difference between the average-rating distribution of the method_cfg and codeGNN models.
- $H3_{null}$: there is a significant difference between the average-rating distribution of the method_only and codeGNN models.

Table 5.4 the result of the significance test. The table shows the p-value, t-static, the decision boundary (α), and the decision made based on p-value.We rejected h1 since the p-value is greater than 0.05, but we

Table 5.2: Example summaries generated by method_cfg

```
public void DisplayEntireMessage(){
   for (int j=0; j < finalMessage.length; j++) {
       finalMessage[j]=String.valueOf(splitMessage[j]);
   }
   amtOfTextShown=message.length();
}
```

Reference: displays the entire message

method_cfg: displays the content of the message as an instance of text

method_only : display the current text in the screen

```
public JSONObject optJSONObject(int index){
    Object o=opt(index);
    return o instanceof JSONObject ? (JSONObject)o : null;
}
```

Reference: get the optional jsonobject associated with an index

method_cfg: overrided by a jsonobject

method_only: get an optional jsonobject associated with a given index

Table 5.3: Summary of average rating for each model

Model	**Mean**	**Median**	**min**	**max**	**Standard Deviation**
method_only	3.42	3.53	2.19	4.50	0.63
method_cfg	3.44	3.44	2.15	4.50	0.65
codeGNN	1.46	1.42	1.23	1.77	0.14

accepted h2 and h3. From this, we can infer that there is no significant difference between the average-rating distribution of the method_cfg and method_only models. On the other hand, there is a significant difference between the average-rating distribution of the method_cfg and codeGNN models. Also, there is a significant difference between the average-rating distribution of the method_only and codeGNN models.

Overall, the result shows method_cfg outperformed the method_only and codeGNN models by the mean average-rating given by experts. Our significance test showed the margin in mean average-rating between method_only and method_cfg models is not statistically significant. On the other hand, both method_cfg and method_only models outperformed codeGNN model by a statistically significant margin of mean average-rating. We conclude method_cfg and method_only are superior.

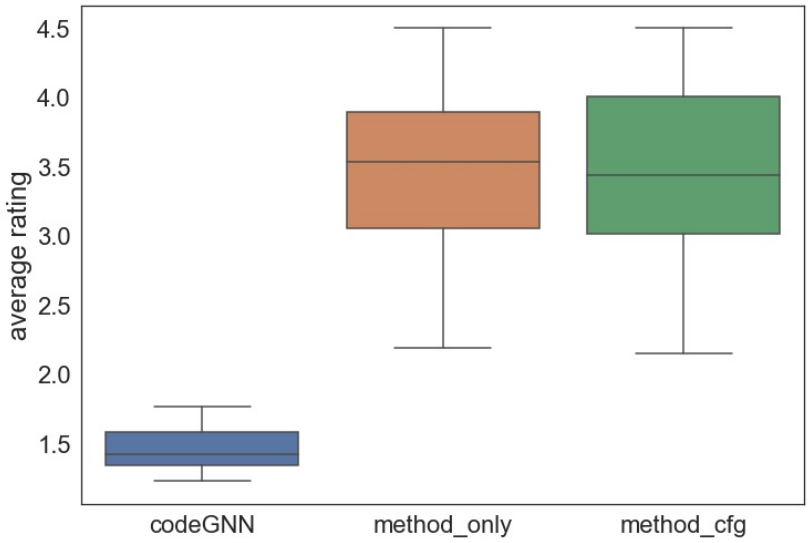

Figure 5.1: Average rating distribution for each model

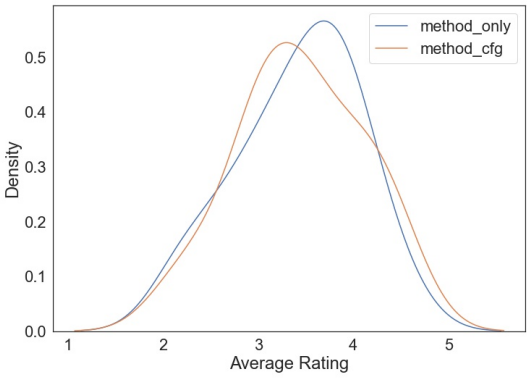

Figure 5.2: Average rating distribution for method_only and method_cfg

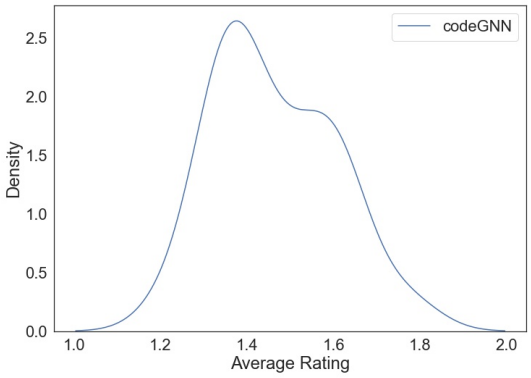

Figure 5.3: Average rating distribution for codeGNN

Table 5.4: Result of statistical significance test

Hypothesis	**p-value**	**t-static**	α	**Decision**
h1	0.92	0.09	0.05	Reject
h2	$< 1e^{-3}$	13.16	0.05	Accept
h3	$< 1e^{-3}$	13.38	0.05	Accept

5.2 Automatic Evaluation

In this study, we compared method_cfg, method_only, and codeGNN model using BLEU score. BLEU measures the n-gram precision of the predicted summaries. In addition, we trained and evaluated the method_only model on a dataset released by LeClair *et al.* [42].

We used 10-fold cross-validation to evaluate the performance of the models. We divided the dataset into training, validation, and testing set on project-level, and this resulted in different training and testing set sizes across different folds. Table 5.5 summarizes the size of training, testing, and validation set on each fold. In addition, Table 5.5 also contains the mean and median sequence length of methods in the test set of each fold.

Table 5.6 shows the 10-fold cross-validation result. We used bold font to highlight the highest BLEU score in each fold. In most of the folds, we found the results of method_cfg and method_only model close to each other while codeGNN is the distant third. The method_only model outperformed method_cfg and codeGNN models in eight out of ten folds. In the remaining two folds (fold three and fold five), the method_cfg model outperformed method_only and codeGNN models. In all except fold one, seven and nine, the method_cfg model outperformed the codeGNN model.

The result of the automatic evaluation shows that encoding linearized control-flow graphs along its method tokens does not improve the prediction accuracy of source-code summarization models in terms of BLEU score. Also, the result shows linearized control-flow graphs are a better alternative than encoding AST using a graph neural network in source-code summarization models.

Also, we trained and evaluated our model (method_only) on the dataset released by LeClair *et al.*[42]. On the dataset, our model scored a BLEU score of 17.53. The performance of the model is inferior to models proposed by Alon *et al.* (code2seq) [4], LeClair *et al.* (AST-attendgru) [41], and LeClair *et al.* (codeGNN) [40] on the dataset. AST-attendgru and codeGNN use AST along with a sequence of method tokens to represent source code, while code2seq uses randomly selected paths between terminal nodes of AST to represent source code. On the other hand, our model got a superior result compared to a model by Iyer *et al.* (codeNN) [32] that uses a sequence of method tokens to represent source code. The summary of the result is shown in Table 5.7. The results of the models shown in the table were taken from an evaluation published by LeClair *et al.* [40].

We attribute the conflict in result to the dataset size used to train the models. Compared to our model, codeGNN is a very large model with 129,180,022 training parameters, while our method_only and method_cfg has 27,083,543 and 39,228,183. Bigger models do not generalize well on small datasets such as ours. Compared to LeClair's dataset with 2.1 million points, our dataset is small, and we expect large models optimized for

Table 5.5: Statistical summary of cross-validation folds

	Size		Method sequence length	
fold	**training set**	**test set**	**mean**	**median**
1	91,553	5,877	13.38	11
2	89,331	8,278	14.37	12
3	90,176	6,465	13.68	12
4	89,660	6,775	13.76	12
5	87,892	8,754	12.71	11
6	91,031	6,255	13.16	11
7	87,759	9,462	13.29	11
8	86,587	10,768	13.08	12
9	75,878	22,706	12.72	11
10	84,915	12,577	12.46	11

large dataset size to perform poorly in small dataset settings. But this does not imply, our dataset size is below the required standard. There is a dataset by Hu *et al.* [29] with 69,708 while ours is 97,917.

Table 5.6: Cross-validation result

fold	**method_only**	**method_cfg**	**codeGNN**
1	**15.42**	12.97	15.35
2	**10.23**	10.12	5.65
3	14.55	**14.91**	8.20
4	**14.03**	12.78	5.14
5	21.49	**21.85**	16.67
6	**11.81**	11.36	5.51
7	**19.26**	13.61	14.48
8	**24.52**	20.17	17.24
9	**39.41**	26.16	38.72
10	**39.92**	35.21	33.42

5.3 Limitations

There are limitations to this work. The first limitation of this work is the dataset used to train and evaluate the models. We were required to use a dataset with compilable projects, so we used a dataset published by

Martins *et al.* [50] to extract the training, validation, and testing set. Because of this, we did not get to set selection criteria for the type of projects included in the dataset. We kept the project containing Javadoc statements that we we were able compile and match their processed methods and summaries with control-flow graphs, everything else is excluded. The project distribution is a power-law distribution, as shown in figure 3.9, where a lot of projects contributed a small number of data points ranging from 1-10. The final dataset and its description can be found using this link.

Second, in creating the dataset, we have applied heuristics to clean and process the dataset. But, we expect noise in both the training and testing set. Also, due to time limitations, we did not verify each data point for correctness, so we acknowledge this as a potential source of error.

Third, we used a 10-fold cross-validation split on the project level to evaluate our results. Due to this, our model (method_cfg) was sensitive to change in data in training and testing across different folds. We chose the hyper-parameters based on the performance on a fold and used the same parameter across 10-folds. Due to this, our result shows the performance of the model on the selected hyper-parameter configuration and data used during training and testing.

Fourth, in the expert evaluation, the participants submitted their evaluation anonymously. Due to this, we couldn't collect data that would enable us to extract trends and behaviors of the human judges both as a group and individually.

Finally, we acknowledge that our approach was not evaluated in other programming languages other than Java. But, the approach we took was general and can be scaled to different programming languages with the availability of data.

Table 5.7: Model performance in BLEU on dataset by LeClair *et al.*

Model	**BLEU score**
method_only	17.53
ast-attendgru	18.69
code2seq	18.84
codeGNN	19.93
codeNN	9.95

5.4 Summary

We showed the result of our automatic and expert evaluation conducted to assess the performance of our model compared to the current state-of-the-art and a baseline that does not use the proposed representation. Our expert evaluation showed as our model (method_cfg) generates better summaries that describe the functionality of a method compared to method_only and codeGNN.

In our automatic evaluation, we saw that our model performance was inferior to method_only a model that does not use the proposed represenation. But, our approach that uses a control-flow graph as a source of structural information performed better from a model that uses AST.

6 Summary

Source-code summarization is a transformation of source code into a natural-language summary. Source-code summarization approaches use information retrieval and natural language processing techniques to generate summaries for source-code artifacts. Based on the type of technique employed, source-code summarization approaches are categorized into three groups: text-retrieval-based, heuristic-based, and data-driven-based source-code summarization. This work comes under data-driven-based source-code summarization.

Data-driven source-code summarization approaches use deep-learning approaches to construct source-code. Approaches under this category differ from one another based on the input representation and model architecture used to design and implement the source-code summarization system. Researchers explored the use of method tokens, parse trees, and AST as input representation. Also, they examined different model architectures that mainly come under sequence-to-sequence models that fit the selected input representation.

This work uses of control-flow graph extracted from the SSA form of a program in source-code summarization models. The control-flow graph is extracted to capture the contextual relationship between blocks of instruction in the SSA form of a program. We encoded the control-flow graph along with method tokens using two bi-directional LSTM encoders and used an LSTM decoder for the summary.

To train and evaluate the model, we created a source-code summarization dataset from a compilable projects dataset by Martins *et al.* [49]. We used a set of heuristics described in chapter 4 (implementations) to process, clean, and tokenize the dataset.

For the experiment, we implemented two models and compared them with the state-of-the-art codeGNN. The first model (method_cfg) is the model that uses control-flow graphs and a sequence of method tokens as an input. The second model (method_only) is the model that only takes a sequence of method tokens as an input. We used a second baseline, the state-of-the-art model (codeGNN) by LeClair *et al.* [40].

We conducted two evaluations: an expert evaluation and an automatic evaluation. In the expert evaluation, we used human judges to rate summaries generated by the three models. In the intrinsic evaluation, we used the BLEU score as a metric to measure the performance of each model on a test set.

In the expert evaluation, the participants of the study gave the highest mean average-rating to the method_cfg model. The second higher mean average-rating was given to the method_only model. The codeGNN model received the least average rating. In a statistical significance test, the difference in mean average-rating between method_cfg and method_only model was not significant. On the other hand, the difference in mean average-rating between both models and codeGNN was significant. Therefore, both of our models significantly outperformed codeGNN.

In the automatic evaluation, we conducted 10-fold cross-validation on a dataset split on the project level. In this evaluation, we used the BLEU score to measure the performance of models. The method_only model outperformed the method_cfg and codeGNN models in 8 out of 10 folds. In the remaining two folds, the method_cfg model got the superior result. In seven out of 10 folds, the method_cfg model outperformed the codeGNN model.

We also trained the method_only model on a dataset published by LeClair *et al.* [42]. We used the same hyperparameter configuration used to train the model in our dataset. The model scored a BLEU score of 17.53 on the dataset. The model did not outperform the state-of-the-art in the dataset, but we got a comparable result. On the dataset, the highest recorded result is a BLEU score of 20.9, and codeGNN model scored a BLEU score of 19.93. The positive side of the method_only model is that it is smaller in size and takes almost half the time required to train the codeGNN model.

In this work, we explored the use of a control-flow graph in a source-code summarization model. Our results show that encoding control-flow graphs along with method tokens does not increase the accuracy of source-code summarization in terms of BLEU score compared to a model that does not use control-flow graphs. In addition, there was no significant difference between the method_cfg and method_only models on the mean average-rating given by the participants of the expert evaluation. On the other hand, we found encoding control-flow graphs along with method tokens is a better alternative in source-code summarization models compared to encoding AST in graph neural nets along with a sequence of method tokens.

Overall, we improved the state-of-the-art in method summarization on our dataset with two models: with and without control-flow graphs. This work also gives an insight into the relation of the BLEU score to human expert evaluations. Also, we constructed a source-code summarization dataset annotated with ASTs and control-flow graphs.

6.1 Future Work

Following this work, we would like to address the limitations of this work. Compared to other datasets, the dataset we used to evaluate our work is medium-sized. We want to evaluate this approach in a larger dataset. In addition, we would like to work on creating a source-code summarization dataset validated for correctness by human judges. Also, we would like to explore other sequence-to-sequence model architectures that can improve the performance of our model. Additionally, we want to work on a larger scale expert evaluation to explore further the BLEU score's interpretation on source-code summarization models. This will give us an insight on the interpretation of the BLEU score in source-code summarization context. Based on the insight, we can further explore evaluation metrics that can be used to evaluate summaries generated for software artifacts. We also would like to include the measure of agreement between our human judges who participated in the evaluation.

6.2 Summary

This work explored the use of structural information extracted from control-flow graphs in a source-code summarization model. To evaluate this approach, we created a source-code summarization dataset annotated with a linearized control-flow graph and ASTs. We conducted human and automatic evaluations. In both evaluations, we did not see a significant improvement due to the use of control-flow graphs in the source-code summarization model compared to a model that does not use a control-flow graph. We improved the state-of-the-art for method-level source-code summarization on our dataset.

Our approaches can be deployed on existing integrated development environments (IDE) either by integrating an inference model that predicts a summary given an input representation or a prediction model that predicts a summary given an input representation and an existing summary. The former can be used to create new summaries for methods, and the latter can be used to update summaries following a code change. But, we expect the model to be trained on large datasets that cover a diverse set of codebases to aid software development and maintenance effectively.

References

[1] N. J. Abid, N. Dragan, M. L. Collard, and J. I. Maletic. Using stereotypes in the automatic generation of natural language summaries for C++ methods. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 561–565, Sep. 2015.

[2] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. A transformer-based approach for source code summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, pages 4998–5007. Association for Computational Linguistics, 2020.

[3] Frances E. Allen. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, page 1–19, New York, NY, USA, 1970. Association for Computing Machinery.

[4] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[5] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[6] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[7] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. Neural code comprehension: A learnable representation of code semantics. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 3589–3601, 2018.

[8] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022, 2003.

[9] BLEU interpretation. `https://cloud.google.com/translate/automl/docs/evaluate`.

[10] Lionel C. Briand. Software documentation: How much is enough? In *7th European Conference on Software Maintenance and Reengineering (CSMR 2003), 26-28 March 2003, Benevento, Italy, Proceedings*, page 13, 2003.

[11] Denny Britz, Anna Goldie, Minh-Thang Luong, and Quoc V. Le. Massive exploration of neural machine translation architectures. *CoRR*, abs/1703.03906, 2017.

[12] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, 2014.

[13] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*, pages 25–35, 1989.

[14] Scott C. Deerwester, Susan T. Dumais, Thomas K. Landauer, George W. Furnas, and Richard A. Harshman. Indexing by latent semantic analysis. *JASIS*, 41(6):391–407, 1990.

[15] Michael J. Denkowski and Alon Lavie. Meteor 1.3: Automatic metric for reliable optimization and evaluation of machine translation systems. In *Proceedings of the Sixth Workshop on Statistical Machine Translation, WMT@EMNLP 2011, Edinburgh, Scotland, UK, July 30-31, 2011*, pages 85–91, 2011.

[16] Natalia Dragan, Michael L Collard, and Jonathan I Maletic. Reverse engineering method stereotypes. In *2006 22nd IEEE International Conference on Software Maintenance*, pages 24–34. IEEE, 2006.

[17] Brian P Eddy, Jeffrey A Robinson, Nicholas A Kraft, and Jeffrey C Carver. Evaluating source code summarization techniques: Replication and expansion. In *21st International Conference on Program Comprehension (ICPC)*, pages 13–22. IEEE, 2013.

[18] Letha H. Etzkorn and Carl G. Davis. Automatically identifying reusable OO legacy code. *IEEE Computer*, 30(10):66–71, 1997.

[19] Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. Structured neural summarization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[20] Beat Fluri, Michael Würsch, and Harald C. Gall. Do code and comments co-evolve? on the relation between source code and comment changes. In *14th Working Conference on Reverse Engineering (WCRE 2007), 28-31 October 2007, Vancouver, BC, Canada*, pages 70–79, 2007.

[21] Github. `https://github.com`.

[22] Sonia Haiduc, Jairo Aponte, and Andrian Marcus. Supporting program comprehension with source code summarization. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pages 223–226, 2010.

[23] Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. On the use of automated text summarization techniques for summarizing source code. In *17th Working Conference on Reverse Engineering, WCRE 2010, 13-16 October 2010, Beverly, MA, USA*, pages 35–44, 2010.

[24] Zellig S Harris. Distributional structure. *Word*, 10(2-3):146–162, 1954.

[25] Emily Hill, Lori L. Pollock, and K. Vijay-Shanker. Automatically capturing source code context of nl-queries for software maintenance and reuse. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, pages 232–242, 2009.

[26] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar T. Devanbu. On the naturalness of software. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 837–847. IEEE Computer Society, 2012.

[27] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[28] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018*, pages 200–210. ACM, 2018.

[29] Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. Summarizing source code with transferred API knowledge. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, pages 2269–2275. ijcai.org, 2018.

[30] Mia Hubert and Stephan Van der Veeken. Outlier detection for skewed data. *Journal of Chemometrics: A Journal of the Chemometrics Society*, 22(3-4):235–246, 2008.

[31] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 2073–2083, 2016.

[32] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics, 2016.

[33] Bernard J. Jansen and Soo Young Rieh. The seventeen theoretical constructs of information searching and information retrieval. *JASIST*, 61(8):1517–1534, 2010.

[34] Karen Sparck Jones. A statistical interpretation of term specificity and its application in retrieval. *J. Documentation*, 60(5):493–502, 2004.

[35] Karen Spärck Jones. Automatic summarising: The state of the art. *Inf. Process. Manage.*, 43(6):1449–1481, 2007.

[36] Mira Kajko-Mattsson. A survey of documentation practice within corrective maintenance. *Empirical Software Engineering*, 10(1):31–55, 2005.

[37] Keras. `https://keras.io/`.

[38] Langdetect. `https://pypi.org/project/langdetect/`.

[39] How to use sparse categorical crossentropy with TensorFlow 2 and Keras? `https://www.machinecurve.com/index.php/2019/10/06/how-to-use-sparse-categorical-crossentropy-in-keras`, 2019.

[40] Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. Improved code summarization via a graph neural network. In *ICPC '20: 28th International Conference on Program Comprehension, Seoul, Republic of Korea, July 13-15, 2020*, pages 184–195. ACM, 2020.

[41] Alexander LeClair, Siyuan Jiang, and Collin McMillan. A neural model for generating natural language summaries of program subroutines. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 795–806. IEEE / ACM, 2019.

[42] Alexander LeClair and Collin McMillan. Recommendations for datasets for source code summarization. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 3931–3937. Association for Computational Linguistics, 2019.

[43] Yuding Liang and Kenny Qili Zhu. Automatic generation of text descriptive comments for code blocks. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, pages 5229–5236. AAAI Press, 2018.

[44] Chin-Yew Lin. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pages 74–81, 2004.

[45] C. Lopes, S. Bajracharya, J. Ossher, and P. Baldi. UCI source code data sets, 2010.

[46] Pablo Loyola, Edison Marrese-Taylor, and Yutaka Matsuo. A neural architecture for generating natural language descriptions from source code changes. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 2: Short Papers*, pages 287–292. Association for Computational Linguistics, 2017.

[47] Yangyang Lu, Zelong Zhao, Ge Li, and Zhi Jin. Learning to generate comments for api-based code snippets. In *Software Engineering and Methodology for Emerging Domains*, pages 3–14, Singapore, 2019. Springer Singapore.

[48] Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, EMNLP 2015, Lisbon, Portugal, September 17-21, 2015*, pages 1412–1421. The Association for Computational Linguistics, 2015.

[49] Pedro Martins, Rohan Achar, and Cristina V. Lopes. 50K-C: a dataset of compilable, and compiled, Java projects. In *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, pages 1–5, 2018.

[50] Pedro Martins, Rohan Achar, and Cristina V Lopes. 50K-C: A dataset of compilable, and compiled, Java projects. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 1–5. IEEE, 2018.

[51] Paul W McBurney and Collin McMillan. Automatic documentation generation via source code summarization of method context. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 279–290. ACM, 2014.

[52] David M. Mimno, Wei Li, and Andrew McCallum. Mixtures of hierarchical topics with pachinko allocation. In *Machine Learning, Proceedings of the Twenty-Fourth International Conference (ICML 2007), Corvallis, Oregon, USA, June 20-24, 2007*, pages 633–640, 2007.

[53] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[54] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 311–318. Association for Computational Linguistics, 2002.

[55] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, July 6-12, 2002, Philadelphia, PA, USA*, pages 311–318, 2002.

[56] Razvan Pascanu, Tomás Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*, volume 28 of *JMLR Workshop and Conference Proceedings*, pages 1310–1318. JMLR.org, 2013.

[57] Sawan Rai, Tejaswini Gaikwad, Sparshi Jain, and Atul Gupta. Method level text summarization for Java code using nano-patterns. In *24th Asia-Pacific Software Engineering Conference, APSEC 2017, Nanjing, China, December 4-8, 2017*, pages 199–208. IEEE Computer Society, 2017.

[58] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. *Learning Representations by Back-Propagating Errors*, page 696–699. MIT Press, Cambridge, MA, USA, 1988.

[59] Gerard Salton and Chris Buckley. Term-weighting approaches in automatic text retrieval. *Inf. Process. Manag.*, 24(5):513–523, 1988.

[60] Gerard Salton, A. Wong, and Chung-Shu Yang. A Vector Space Model for Automatic Indexing. *Commun. ACM*, 18(11):613–620, 1975.

[61] Janice Singer, Timothy C. Lethbridge, Norman G. Vinson, and Nicolas Anquetil. An examination of software engineering work practices. In *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative Research, November 10-13, 1997, Toronto, Ontario, Canada*, page 21, 1997.

[62] Jeremy Singer, Gavin Brown, Mikel Luján, Adam Pocock, and Paraskevas Yiapanis. Fundamental nano-patterns to characterize and classify Java methods. *Electronic Notes in Theoretical Computer Science*, 253(7):191–204, 2010. Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications (LDTA 2009).

[63] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. Towards automatically generating summary comments for Java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 43–52. ACM, 2010.

[64] Josef Steinberger and Karel Jezek. Update summarization based on latent semantic analysis. In *Text, Speech and Dialogue, 12th International Conference, TSD 2009, Pilsen, Czech Republic, September 13-17, 2009. Proceedings*, volume 5729 of *Lecture Notes in Computer Science*, pages 77–84. Springer, 2009.

[65] Ilya Sutskever, James Martens, and Geoffrey E. Hinton. Generating text with recurrent neural networks. In *Proceedings of the 28th International Conference on Machine Learning, ICML 2011, Bellevue, Washington, USA, June 28 - July 2, 2011*, pages 1017–1024. Omnipress, 2011.

[66] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, pages 3104–3112, 2014.

[67] Ted Tenny. Program readability: Procedures versus comments. *IEEE Trans. Software Eng.*, 14(9):1271–1279, 1988.

[68] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research, November 8-11, 1999, Mississauga, Ontario, Canada*, page 13. IBM, 1999.

[69] Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. Code generation as a dual task of code summarization. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*, pages 6559–6569, 2019.

[70] P.J. Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[71] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI Open*, 1:57–81, 2020.

[72] Yuxiang Zhu and Minxue Pan. Automatic code summarization: A systematic literature review. *CoRR*, abs/1909.04352, 2019.

Appendix A

Appendix

A.1 Generated Summaries

In this section, we show example methods and summaries predicted by our models and the current state-of-the-art.

```
public Vector2D mul (double a) {
   this.x *= a;
   this.y *= a;
   return this;
}
```

Reference: multiply this vector by the given scalar
method_only: in place in this vector by a scalar
method_cfg: multiplies the vector with the given scalar
codeGNN: returns the value of the

```
public int getDepth(boolean inclusive){
    int depth;
    if (inclusive) depth=1;
    else  depth=0;
    if (parent != null) return depth + parent.getDepth(true);
    else  return depth;
}
```

Reference: get the length of the path from this node to the root
method_only:returns the depth of the operand in this stack
method_cfg: returns the depth of the node in the search report
codeGNN: returns the the of the

```
private static String getCurrentTimeStamp(){
    Calendar cal = Calendar.getInstance();
    SimpleDateFormat sdfDate = new SimpleDateFormat("yyyyMMdd_HHmmss");
    String date = sdfDate.format(cal.getTime());
    return date;}
```

Reference: get the current time stamp from the running computer
method_only:gets the current time stamp
method_cfg: returns the current time in the format in the format
codeGNN: returns the the of the

```
private Date getRandomDate(){
    return new Date(System.currentTimeMillis() + (long)86400000 * random.nextInt(DATE_SPAN_YEARS *
        365));
}
```

Reference: picks random date from date span years interval
method_only:returns a random date in the given date

method_cfg: returns a random date in the time and returns it
codeGNN: returns the the of the

A.2 Expert Evaluation Results

In this section, we show the raw data of the user study in table 1.1. The table contains the evaluation of 20methods used in the study. We show the 5-level Likert scale result of all models along with the total and average score of each model. To compute the total score, we the number of ratings by the corresponding rating score. For example, we multiply the number of users who strongly disagreed with the summary by 1, the number of users who disagreed with the summary by 2, and so on. Then, we add the numbers to compute the total value. Finally, we divide the total with the number of participants, 26, to compute the average rating.

Table A.1: Expert evaluation raw data

Method	Model	Strongly disagree	Disagree	Neutral	Agree	Strongly agree	Total	Average
Method 1	codeGNN	14	8	4	0	0	42	1.615384615
	method_only	1	6	5	10	4	88	3.384615385
	method_cfg	0	7	8	9	2	84	3.230769231
Method 2	codeGNN	15	8	3	0	0	40	1.538461538
	method_only	2	5	7	10	2	83	3.192307692
	method_cfg	2	8	8	7	1	75	2.884615385
Method 3	codeGNN	20	2	4	0	0	36	1.384615385
	method_only	2	2	2	10	10	102	3.923076923
	method_cfg	4	2	2	10	8	94	3.615384615
Method 4	codeGNN	19	4	3	0	0	36	1.384615385
	method_only	0	0	0	13	13	117	4.5
	method_cfg	0	0	0	14	12	116	4.461538462
Method 5	codeGNN	13	10	3	0	0	42	1.615384615
	method_only	0	1	4	18	3	101	3.884615385
	method_cfg	5	12	5	3	1	61	2.346153846
Method 6	codeGNN	18	7	1	0	0	35	1.346153846
	method_only	0	1	2	13	10	110	4.230769231
	method_cfg	0	2	1	12	11	110	4.230769231
Method 7	codeGNN	19	5	2	0	0	35	1.346153846
	method_only	1	8	4	10	3	84	3.230769231
	method_cfg	0	6	5	12	3	90	3.461538462
Method 8	codeGNN	15	7	3	1	0	42	1.615384615
	method_only	1	2	5	14	4	96	3.692307692
	method_cfg	3	1	5	11	6	94	3.615384615
Method 9	codeGNN	19	5	2	0	0	35	1.346153846
	method_only	5	4	10	7	0	71	2.730769231
	method_cfg	1	11	7	6	1	73	2.807692308
Method 10	codeGNN	16	8	1	1	0	39	1.5
	method_only	1	5	5	11	4	90	3.461538462
	method_cfg	0	6	13	4	3	82	3.153846154
Method 11	codeGNN	15	8	2	1	0	41	1.576923077
	method_only	0	1	7	14	4	99	3.807692308
	method_cfg	2	8	7	7	2	77	2.961538462
Method 12	codeGNN	17	7	2	0	0	37	1.423076923
	method_only	0	2	3	12	9	106	4.076923077
	method_cfg	1	1	13	8	3	89	3.423076923
Method 13	codeGNN	18	5	3	0	0	37	1.423076923
	method_only	1	1	9	11	4	94	3.615384615
	method_cfg	1	2	8	11	4	93	3.576923077
Method 14	codeGNN	19	5	2	0	0	35	1.346153846
	method_only	1	3	3	9	10	102	3.923076923
	method_cfg	1	2	2	12	9	104	4

Method 15	codeGNN	13	10	3	0	0	42	1.615384615
	method_only	2	4	12	5	3	81	3.115384615
	method_cfg	1	0	0	18	7	108	4.153846154
Method 16	codeGNN	19	5	2	0	0	35	1.346153846
	method_only	0	1	8	15	2	96	3.692307692
	method_cfg	0	0	2	9	15	117	4.5
Method 17	codeGNN	21	4	1	0	0	32	1.230769231
	method_only	2	8	14	1	1	69	2.653846154
	method_cfg	0	2	3	13	8	105	4.038461538
Method 18	codeGNN	18	7	1	0	0	35	1.346153846
	method_only	1	9	10	4	2	75	2.884615385
	method_cfg	1	7	6	11	1	82	3.153846154
Method 19	codeGNN	10	12	4	0	0	46	1.769230769
	method_only	5	12	8	1	0	57	2.192307692
	method_cfg	4	6	3	11	2	79	3.038461538
Method 20	codeGNN	14	11	1	0	0	39	1.5
	method_only	7	10	5	4	0	58	2.230769231
	method_cfg	8	10	4	4	0	56	2.153846154

A.3 Expert Evaluation User Interface

In this section, we present two figures that show the user interface used in the expert evaluation. The two figures show the evaluation page of the expert-evaluation user interface. Figure A.1 show the method and summaries generated by each model. Figure A.2 show the radio buttons used to select rating for each summary. Also, in figure A.2 we can also see an optional field to submit a summary for the given method.

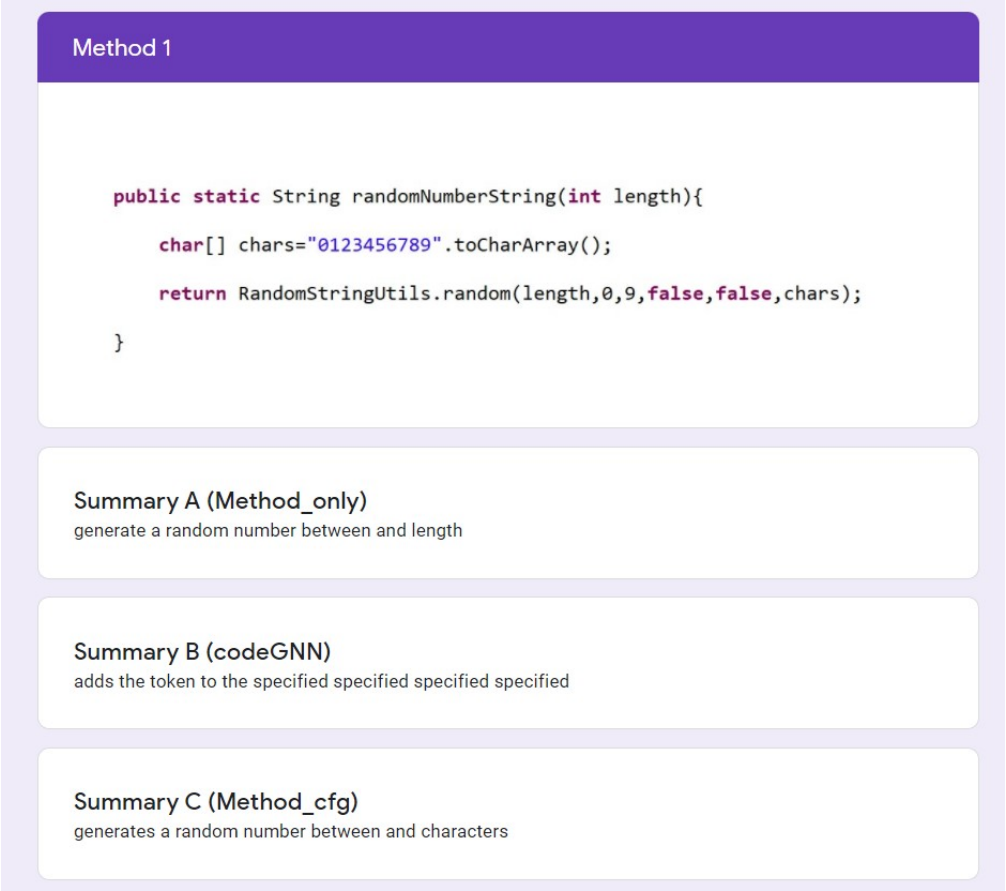

Figure A.1: Expert evaluation user interface 1

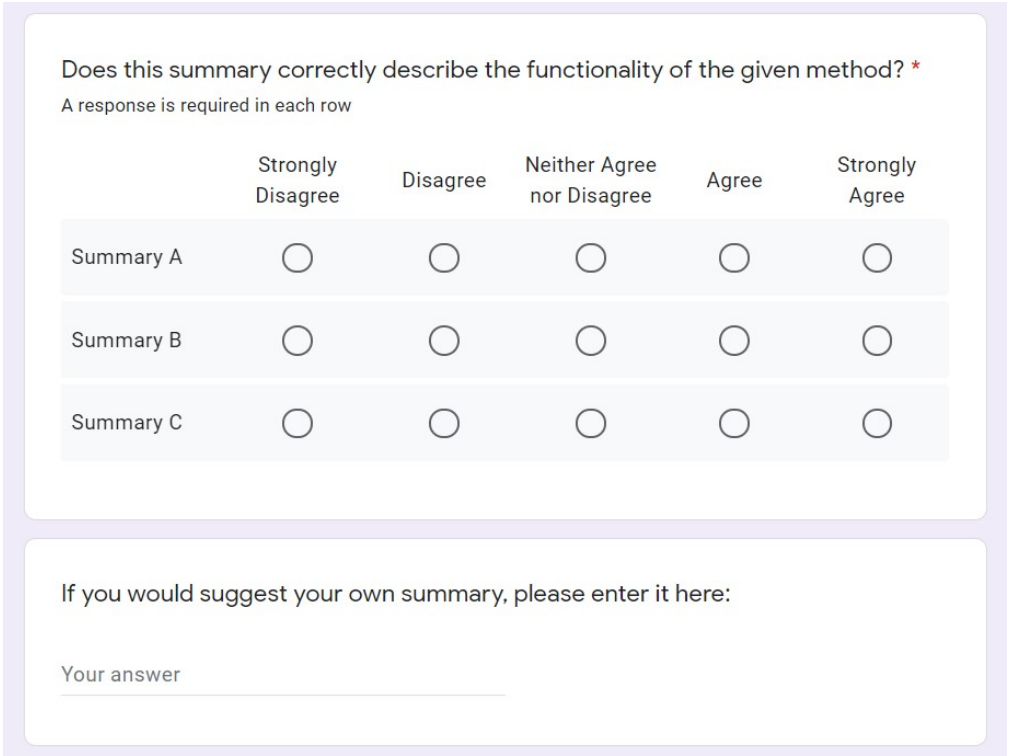

Figure A.2: Expert evaluation user interface 2